MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un debugger Prolog

van Rossum, Emmanuel

Award date:
1989

Awarding institution:
Universite de Namur

Facultés Universitaires Notre Dame de la Paix à Namur

Implémentation d'un debugger Prolog

Emmanuel van Rossum

Mémoire présenté en vue de l'obtention du titre de licencié et maître en informatique

Promoteur: Baudouin Le Charlier

Année académique 1988 - 1989

Remerciements

Je tiens tout d'abord à remercier Pascal Van Hentenryck qui a rendu possible mon stage à l'ECRC, m'a aidé à aborder les implémentations de Prolog et conseillé dans l'élaboration de mon mémoire.

Mes remerciements vont également à tous les chercheurs de l'ECRC qui ont accueilli mes questions avec gentillesse et efficacité. Je pense entre autres à Abder Aggoun, Mireille Ducassé, Anna Emde, Thomas Graf, Joachim Schimpf et plus particulièrement à Micha Meier.

Enfin, je tiens aussi à remercier monsieur Baudouin Le Charlier qui m'a suivi avec beaucoup de sérieux dans la rédaction de ce mémoire et m'a permis d'aborder le sujet très intéressant qu'est la conception et l'implémentation d'un debugger Prolog ainsi que messieurs Hervé Gallaire et Alexander Herold, qui m'ont accueilli au sein de l'équipe "logic programming" à l'ECRC pendant les 6 mois de mon stage.

Table des matières

Abstract

This work follows a stage accomplished in ECRC (European Computer-Industry Research Center). I have implemented there a debugger on a compiled Prolog named SEPIA which stand for 'Standard ECRC Prolog Integrating Advanced features'.

The aim of a debugger is to help the programmer finding and understanding the errors contained in his program. This needs a precise way of tracing program execution. Therefore, it is preferable that the trace supplied by the debugger follows a model independent from the implementation. On the other hand, it is also important for the model to be sufficiently connected with Prolog execution (like the procedural way of Prolog execution).

The more the model is disconnected from the concerned Prolog implementation, the more the problems are serious to solve when implementing the debugger.

Section 1 first reports the minimum Prolog implementation and its optimisations then introduce the Warren abstract machine [Warren 83] for a compiled Prolog. Section 2 introduces Byrd's model for tracing Prolog programs [Byrd 80] and presents a way to extend this model. Finally section 3 develops different problems that occur when implementing a debugger on a Warren abstract machine and the solutions that may be used.

Sommaire

Le présent mémoire est le résultat d'un stage effectué à l'ECRC (European Computer-Industry Research Center) à Munich où j'ai implémenté un debugger sur un Prolog compilé existant: SEPIA (Standard ECRC Prolog Integrating Advanced features).

Un debugger (outil d'aide à la correction de programmes) doit s'attacher à décrire de la manière la plus précise possible l'exécution d'un programme ou d'une partie de programme afin de faciliter le travail de l'utilisateur dans sa tâche de localisation et de compréhention des erreurs que contient son programme.

Il est, dans ces conditions, préférable qu'un debugger trace l'exécution d'un programme dans un modèle indépendant d'une implémentation particulière de Prolog (et donc forcément de ses optimisations) mais néanmoins suffisamment précis, tenant compte par exemple de l'aspect procédural de ce language.

L'implémentation d'un tel debugger nécessite la résolution des problèmes dus au "gap" existant entre le modèle et l'implémentation de Prolog, ce "gap" est d'autant plus important que le système Prolog est optimisé.

La section 1 décrit une implémentation minimum de Prolog ainsi que les optimisations les plus courantes; puis, décrit la machine abstraite de Warren [Warren 83] (adaptée à la compilation de Prolog). La section 2 décrit un modèle de trace procédural en partant du modèle créé par Byrd [Byrd 80]; puis étend celui-ci. Quelques outils de trace sont aussi introduits. Enfin, la section 3 décrit les problèmes et les solutions qui interviennent dans la mise en oeuvre d'un debugger dans le cadre d'une machine abstraite de Warren.

Introduction

Le Prolog SEPIA développé à l'ECRC offre à l'utilisateur, outre les fonctions standard des Prolog courants, des extentions particulières tel qu'un mécanisme de mise en attente de procédure lorsqu'une condition particulière n'est pas remplie, une structure de bloc, la gestion de modules et un système d'interruption.

Le debugger de SEPIA que j'ai écrit lors de mon stage à l'ECRC tient compte de ces particularités et offre des outils qui aident à leur utilisation comme par exemple signaler le programmeur lorsqu'une procédure est mise en attente, signaler lorsqu'elle est "réveillée", signaler la sortie impérative d'un bloc, respecter les modules,...

Ma première tâche fut de comprendre le mode d'implémentation d'un système Prolog d'abord interprèté (car il est plus simple à comprendre) ensuite compilé (c'est le cas de SEPIA). La section 1 de ce rapport décrit donc une implémentation minimum d'un interprétteur Prolog et des optimisations possibles (elles sont faites dans SEPIA). Ensuite je présente la machine abstraite de Warren qui est la base des systèmes Prolog compilés. Bien que SEPIA soit également basé sur cette machine, il comporte une série de différences. Ces différences ne seront pas introduites dans ce rapport, autant pour des raisons de confidentialités que pour des raisons de volume. La présentation de l'implémentation de Prolog est indispensable pour comprendre d'une part le modèle de trace basé sur l'implémentation et d'autre part, les problèmes que les optimisations peuvent poser lors de l'implémentation d'un débugger.

Aucune spécification de debugger n'ayant encore été faite lors de mon arrivée à l'ECRC, j'ai analysé les debuggers existant dans la litérature (modèle de Byrd) et dans les principales implémentations de Prolog disponibles à l'ECRC (Cprolog, Quintus Prolog, Sixtus Prolog, Muprolog, Siemens Prolog et BIM Prolog). Les traces qu'ils mettent en oeuvre peuvent être classées globalement en deux catégories. Celles davantage basées sur un vision de l'exécution proche de l'implémentation et celles basées sur un modèle plus abstrait: le modèle de la boîte inventé par Byrd. Je présente ces deux modèles dans la section 2 et les critique. Ensuite je présente une manière de faire évoluer le modèle de Byrd vers un modèle qui tient compte de ces critiques. Enfin j'analyse le problème de l'affichage des noms des variables et de leurs instantiations ainsi que quelques outils de trace.

La section 3 explique les différents problèmes que l'implémentation d'un debugger comporte dans le cadre d'une machine abstraite de Warren. Ils sont pour la plupart dus au fait que le modèle de trace est éloigné de l'implémentation. Les problèmes sont d'autant plus importants que la machine abstraite de Warren effectue des optimisations. Si l'on décide de supprimer les optimisations lors du debugging, la plupart des problèmes disparaissent. Dans la mesure où l'équipe de SEPIA désirait modifier le moins possible la machine abstraite de SEPIA, il a fallu mettre en oeuvre des solutions adéquates. J'analyse dans la section 3 les problèmes et les solutions à mettre en oeuvre dans de telles conditions.

1. Prolog et son implémentation

1. Le prolog de base

Il est supposé que le lecteur ait une connaissance de Prolog au moins en tant qu'utilisateur. Voici une description succinte du langage Prolog. Le but de cette description est essentiellement d'avoir un support terminologique pour décrire son implémentation et les différentes manières de tracer l'exécution d'un programme.

Un **terme** peut être une variable, une constante, une liste, ou une structure.

Un **argument** peut être n'importe quel terme.

Un terme **s'unifie** à un autre terme s'il existe une substitution des variables contenues dans ces termes tel que les termes sont égaux après substitution.

L'**arité** est le nombre d'arguments d'une fonction ou d'une procédure.

Un **foncteur** est un symbole représentant le nom d'une structure ou d'une procédure (c'est le couple foncteur/arité qui permet d'identifier une procédure).

Un **but** est formé d'un foncteur et d'une liste d'arguments.Une **clause** Prolog est formée d'un but (ou tête de la clause) et d'une liste éventuellement vide de sous-buts (le corps de la clause).

Un **fait** est une clause sans sous-buts.

Une **procédure** Prolog est une liste de clauses dont les têtes de clauses ont même foncteur et même arité.

Un **programme** Prolog est un ensemble de procédures.

2. Implémentation minimum de Prolog

Un programme Prolog est toujours exécuté en relation avec une liste non vide de buts. Il s'agit d'essayer de prouver, à l'aide des procédures du programme, que tous les buts de cette liste sont vrais (et de trouver les valeurs des variables contenues dans cette liste qui permettent de le prouver).

Au fur et à mesure que l'exécution du programme évolue, le système va créer un arbre dans lequel chaque noeud représente un sous-but à prouver ainsi qu'une série d'informations qu'il doit conserver pour la suite de l'exécution. Un moment donné de l'exécution peut donc être représenté par l'état de l'arbre à ce moment.

1. La représentation des variables

Suite aux unifications, des variables (contenues dans les arguments des buts) sont liées à des termes éventuellement structurés.

Une variable est **liée** lorsque, suite à une unification, sa valeur est **instanciée** à un terme. Sa valeur dépend alors de ce terme. Lorsque la variable n'est pas liée, on dit qu'elle est **libre**.

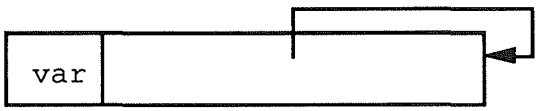
Par exemple, lorsque l'on essaye de prouver `p(X)`, la variable `X` est libre mais une fois que `p(X)` a été unifié à la tête d'une clause soit `p(a)`, on dit que la variable `X` est liée à la constante `a`.

Une variable libre est représentée par un **tag** et une **valeur**. Le tag spécifie à quel type de terme la variable est liée et la valeur contient une instance de ce type.

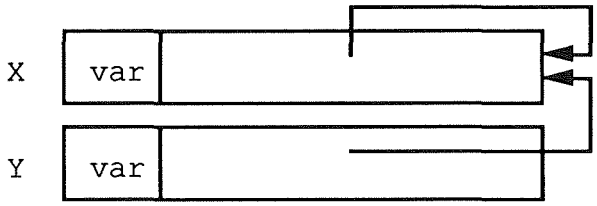
Par exemple, Si la variable `X` est liée à la constante `a` nous aurons:

Si la variable `X` est liée à l'entier `123`, nous aurons:

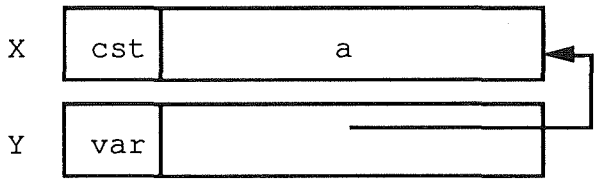
Généralement, une variable libre est représentée comme étant liée à elle-même:



Imaginons le scénario suivant: une variable `X` est créée, une variable `Y` est créée, la variable `X` est unifiée à la variable `Y`:



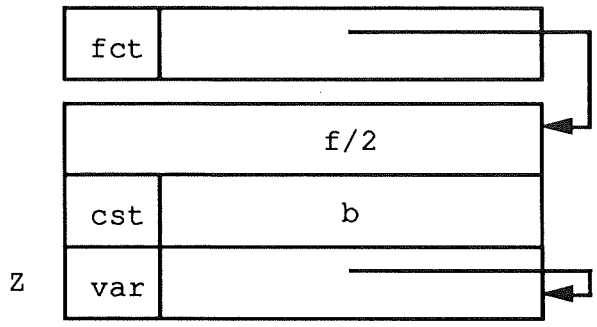
puis la variable `Y` est liée à la constante a. Nous aurons alors la situation suivante:



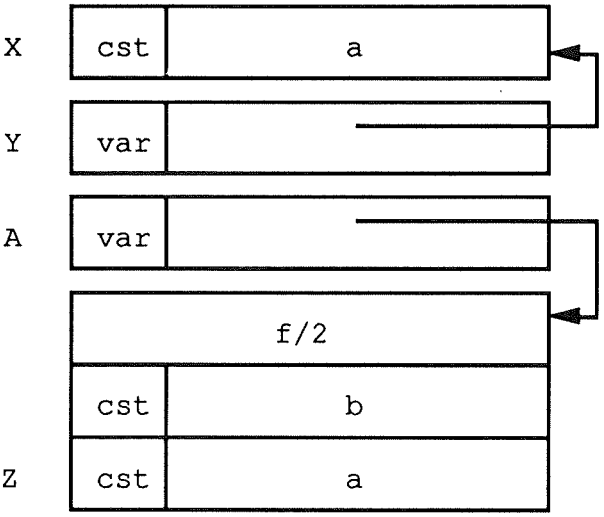
Remarquons que la variable la plus jeune (`Y`) a été liée à la variable la plus ancienne (`X`) et non pas le contraire. A ce stade-ci, nous aurions pu implémenter le contraire, mais nous verrons plus tard que cela aurait empêché certaines optimisations.

En ce qui concerne les termes structurés, il existe deux méthodes de représentation: la représentation par recopie de structure et la représentation par partage de structure. Nous allons introduire ici une des deux méthodes: la recopie de structure car c'est celle qui est utilisée dans la machine abstraite de Warren (introduite plus loin).

Une structure est représentée par son foncteur et son arité suivie des termes qui la composent. Par exemple, la strucure `f(b,Z)`, où `Z` est libre, sera représentée telle que:

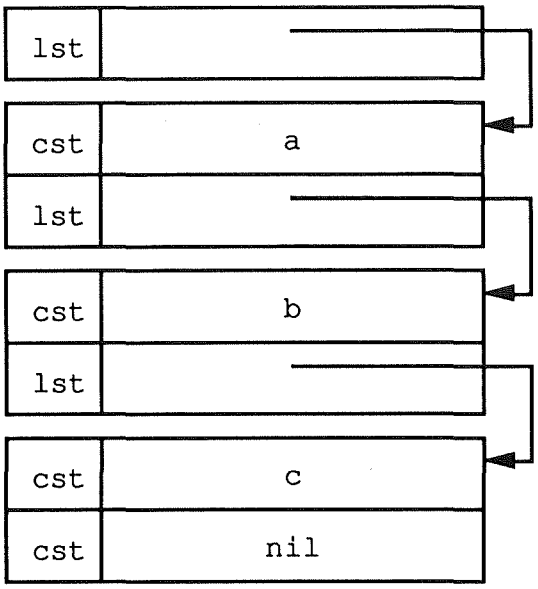


Continuons le scénario: une variable `A` est créée, la variable A est instanciée au terme fonctionnel `f(a,Z)` et `Z` est instancié à `Y`. Nous obtenons alors:



Lorsqu'une variable est accédée, elle est **déréférencée**. La **déréférence** d'une variable `X` est le couple (tag,valeur) qui représente la valeur de la variable du bout de la chaine de référence (le tag de la déréférence ne peut donc être '`var`' que si la déréférence n'est pas liée). C'est la raison pour laquelle la variable `Z` à été liée à la constante `a` (qui est la déréférenciation de la variable `Y`) et non pas à la variable `Y`.

Les termes structurés de type liste sont représentés de manière similaire. La différence est qu'il n'y a pas de foncteur et la liste est composée de deux éléments: une tête et une queue. La liste [a,b,c] est représentée par:



2. Les informations qui caractérisent l'état de l'exécution

Les buts que l'on tente de prouver vont l'être à l'aide du texte du programme (un ensemble de procédures). Plusieurs clauses peuvent servir à prouver un but et l'on ne pourra être sûr que ce but particulier n'est pas prouvable que si l'on a essayé toutes les clauses dont la tête a le même foncteur et le même nombre d'arguments que le but à prouver. Il est donc logique qu'un noeud de l'arbre enregistre une référence vers la clause qui est couramment utilisée pour prouver ce but. Ou, ce qui revient au même, il faut qu'il garde une référence vers la prochaine clause qu'il peut essayer. S'il n'y a plus de telle clause, on dit que la procédure n'est plus un **point de choix**.

Une clause du programme est utilisée pour la preuve lorsque sa tête s'unifie avec le but courant à prouver. Si cette clause est utilisée, il faut allouer de la place pour les variables de la clause. Le noeud va donc aussi contenir les variables de la clause.L'ensemble de ces variables s'appelle un **environnement**. Les termes structurés seront créés sur une pile séparée: **la pile de recopie**.

Lorsqu'aucune clause ne convient pour un but à prouver, il faut faire un **rétro-parcours** pour essayer de trouver un point de choix. En fait, ce que l'on veut c'est recommencer l'exécution avec exactement le même état de l'exécution que celui que l'on avait lors du dernier point de choix mais de recommencer avec la clause suivante. Pour cela, il ne suffit pas de détruire toutes les branches qui ont été créées depuis lors, il faut aussi restituer les variables dans l'état dans lequel elles étaient lors de ce point de choix. Or l'unification a lié certaines variables aux variables de la clause. Dès lors, pour être capable de défaire les liens qui ont été faits, une référence vers chaque variable que l'on lie est enregistrée dans ce que l'on appelle le **trail** (qui fonctionne comme une pile). Ainsi, lors du rétro-parcours, on rend à nouveau libre (on **untrail** en dépilant) les variables qui ont été liées depuis le dernier point de choix. Cela implique que la position du trail soit également enregistée dans les noeuds de l'arbre qui sont des points de choix.

Afin de prouver un but à l'aide d'une clause donnée, tous les sous-buts de cette clause doivent être prouvés. Il est donc logique que le noeud garde une référence vers le sous-but qu'il doit prouver (dans la clause qu'il utilise pour l'instant). Cette information sera aussi indispensable lors du rétro-parcours pour savoir quel était le but à prouver dans cet état du système. Cette information s'appelle la **continuation**.

Afin de pouvoir représenter l'arbre d'exécution sous forme de pile (que l'on appelle la **pile des environnements** ou **pile locale**), nous allons aussi sauver dans chaque noeud une référence (sur la pile) vers son noeud père.

Résumons les différentes informations que chaque noeud doit conserver:

- une référence vers le but que l'on essaye de prouver (dans la clause courante),
- les valeurs et les liens des variables de la clause (l'environnement),
- une référence vers le noeud père.

Et si le noeud est un point de choix, il faut aussi conserver:

- une référence vers la prochaine clause à essayer,
- le sommet de la pile de trail,
- le sommet de la pile de recopie,
- une référence vers le noeud du point de choix précédent afin de trouver plus rapidement le point de choix le plus récent lors d'un rétro-parcours.

Nous avons aussi besoin d'informations globales sur l'état de l'arbre (c'est-à dire de l'exécution):

- une référence vers le noeud en cours de développement (ou plutôt vers son environnement): `EnvCourant`.
- une référence vers le but en cours d'analyse (référence vers le texte du programme): `ButCourant`.
- une référence vers le noeud du dernier point de choix: `DernierPc`.

3. L'algorithme d'exécution

Ce qui suit est la description de ce qu'un système prolog doit accomplir pour résoudre une liste de buts `Bs`.

La pile d'environnements est initialisée avec un environnement dont le but et l'environnement père sont nil.

`PROUVER` est appelé avec `DernierPc` valant nil, `EnvCourant` pointant sur l'environnement au sommet de la pile d'environnements, `ButCourant` pointant sur le premier sous-but de `Bs` et `ClauseCourante` pointant sur la première clause du texte de la procédure qui a le même foncteur que le `ButCourant`.

De manière intuitive, `PROUVER` consiste à essayer les clauses du programme une à une; chaque fois que l'une d'entre-elles peut s'appliquer au `ButCourant` on l'utilise; si elle ne peut s'appliquer on cherche une alternative (point de choix).

```
PROUVER consiste à répéter:
   * (CALL[1] du ButCourant: créer une nouvelle
     branche à partir du noeud courant, il s'agit de
     l'essai d'un but.)
   • s'il existe une clause Cl à partir de la
     ClauseCourante dans le texte du programme dont
     la tête a même foncteur et même arité que le
     ButCourant et que l'unification de la tête de
     cette clause avec le ButCourant réussit (il
     faut mémoriser les liens qui sont créés lors de
     l'unification et il faut trailer les variables
     liées)
      | AVANCER
   • sinon (FAIL du ButCourant)
      | RETRO-PARCOURS
```

De manière intuitive, AVANCER consiste à utiliser la clause Cl pour prouver le ButCourant. Si cette clause n'a pas de sous-but, le but courant est prouvé et il faut remonter dans l'arbre vers le prochain but à prouver (en utilisant les continuations sauvées); si la clause a des sous-buts, il faut commencer par prouver le premier d'entre eux.

[1] Les mots CALL, EXIT, REDO et FAIL seront introduits à la section suivante. Il sont déja signalés ici pour faciliter le parallèle avec les modèles de trace plus tard.

AVANCER consiste à
- faire pointer ClauseCourante sur la clause Cl
- s'il existe encore une clause avec même foncteur et même arité
  - sauver sur la pile les informations d'un point de choix
  - faire pointer DernierPc sur ce noeud
- créer un environnement sur la pile et y sauver les informations nécessaires:
  - le ButCourant
  - les valeurs des variables de la clause (suite à l'unification)
  - EnvCourant (c. à d. le père du nouvel environnement)
- faire pointer EnvCourant sur le nouvel environnement créé
- faire pointer ButCourant sur le premier but de la ClauseCourante (nil s'il n'y en a pas)
- tant que le ButCourant est nil
  - si l'EnvCourant est nil, fin de l'exécution, il s'agit d'un succès
  - (**EXIT du père du ButCourant**: remonter le long de la branche qui mène au noeud courant: tous les sous-buts du père ont réussi, le père a donc aussi réussi)
  - faire pointer EnvCourant sur son père
  - faire pointer ButCourant sur le but qui suit (dans le texte du programme) le but de EnvCourant
- faire pointer ClauseCourante sur la première clause du texte de la procédure qui a même foncteur et même arité que le but courant

De manière intuitive, RETRO-PARCOURS consiste à se remettre dans l'état de l'exécution du dernier point de choix et à pointer vers la clause suivante de ce point de choix.

RETRO-PARCOURS consiste à:
- si DernierPc = nil
  - il n'y a plus d'alternative, Bs ne peut être prouvé à l'aide du programme.
- sinon
  - * (faire un retro-parcours: toutes les branches créées depuis le dernier point de choix sont détruites. On récupère alors l'arbre qui correspondait à l'état de l'exécution au moment du CALL du point de choix et l'on reprend l'exécution mais avec la clause suivante de ce point de choix, **le prochain CALL = REDO**)
  - faire pointer ClauseCourante sur la Clause sauvée dans le point de choix pointé par DernierPc
  - faire pointer ButCourant sur le But sauvé dans l'environnement pointé par DernierPc
  - faire pointer EnvCourant sur l'environnement pointé par DernierPc
  - toutes les variables notées dans le trail jusqu'à la valeur du trail sauvée dans le point de choix pointé par DernierPc, sont rendues libres (le trail est "dépilé" jusqu'à cette valeur)
  - faire pointer DernierPc sur le point de choix pointé par le pointeur sauvé dans le point de choix pointé par DernierPC
  - la pile de prolog est détruite jusqu'à l'EnvCourant (celui-ci devient le sommet de la pile Prolog)

3. Le cut

Le cut peut être considéré comme un sous-but d'une clause qui réussit toujours et qui a en plus un effet de bord: tous les points de choix qui ont été créés depuis le but qui a le cut comme sous-but, ne peuvent plus être considérés comme des points de choix.

Du point de vue de l'implémentation, le principe est simple, il faut faire passer à nil la référence vers la clause suivante de tous les points de choix qui ont été empilés sur la pile depuis le CALL du père du cut.

En supposant qu'il existe un fait 'cut' prédéfini, il nous suffit de modifier l'algorithme précédent en remplacant AVANCER par AVANCER' tel que:

```
AVANCER' consiste à
  • si le ButCourant est cut
      • pour tous les environnements Env à partir
        du sommet de la pile jusqu'à l'EnvCourant
          • si l'environnement Env est un point de
            choix
              • mettre nil comme clause suivante de
                ce point de choix (donc ce n'est
                plus un point de choix)
      • tant que le ButCourant est nil
          • si l'EnvCourant est nil, fin de
            l'exécution, il s'agit d'un succès
          * (EXIT du père du ButCourant)
            remonter le long de la branche qui mène
            au noeud courant: tous les sous-buts du
            père ont réussi, le père a donc aussi
            réussi
          • faire pointer EnvCourant sur son père
          • faire pointer ButCourant sur le but qui
            suit (dans le texte du programme) le but
            de EnvCourant
  • sinon
      • AVANCER
```

6. Les optimisations

Cette section, ainsi que la suivante, concerne des considérations d'implémentations qu'il n'est pas nécessaire de comprendre pour aborder la section concernant les spécifications fonctionnelles du debugger; elles le sont néanmoins pour les spécifications d'implémentation du debugger de Sepia qui met en oeuvre ces optimisations (section 1.6) dans une machine abstraite Prolog (section 1.7).

1. Les sous-arbres déterministes

Dans la suite, nous qualifierons de **sous-arbre déterministe**, un sous-arbre de l'arbre ET de l'exécution qui ne contient aucun point de choix.

Un sous-arbre déterministe est donc, par définition, un sous-arbre dans lequel l'exécution ne fera plus jamais de rétro-parcours. L'idée de l'optimisation des sous-arbres déterministes est donc de ne pas conserver les informations d'un tel sous-arbre lorsque l'on en sort.

Cela n'est en fait pas toujours possible. Pour pouvoir supprimer un noeud de l'arbre il ne suffit pas d'être sûr que l'exécution n'y reviendra plus lors d'un rétro-parcours, il faut encore que ce noeud ne contienne aucun terme que la suite de l'exécution utilisera encore. En d'autres termes, il ne faut pas qu'il y ait de variable plus ancienne que ce noeud qui soit liée à un terme de ce noeud.

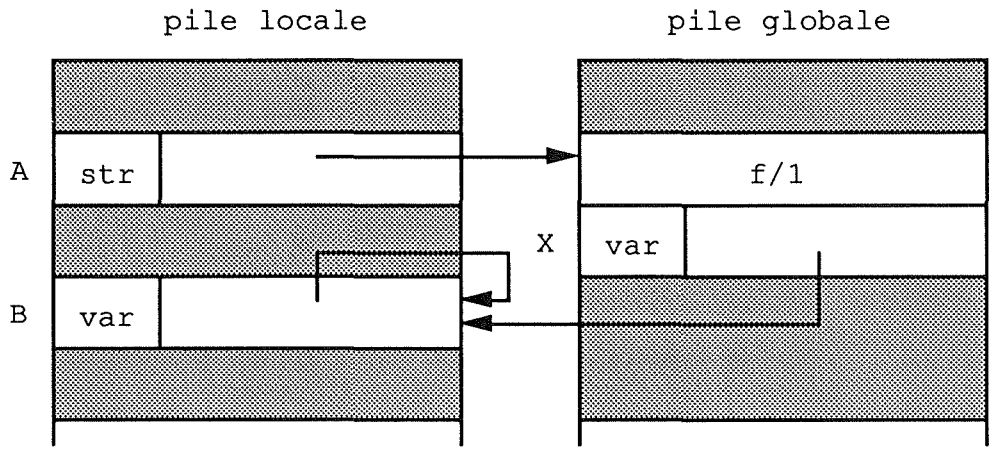
Pour éviter une telle situation (et être alors toujours à même de supprimer un noeud lorsqu'il s'avère être deterministe), il faut s'assurer qu'une variable n'est jamais liée à un terme stocké dans un noeud plus récent (c'est-à dire créé plus tard) que le noeud auquel elle appartient. J'appelerai cette règle **la règle de chronologie**.

Soit une variable `A` d'un noeud `N` plus ancienne que la variable `B` du noeud `M`. Nous avons vu plus haut que lorsqu'une variable s'unifie à une autre, la liaison se faisait toujours de la plus jeune vers la plus vieille. Cela nous permet de respecter la règle de chronologie.

Soit une variable A d'un noeud N plus ancienne qu'une strucure S d'un noeud M. S'il faut lier la variable A à la structure S, nous aurons une variable liée à un terme structuré plus jeune. Néanmoins, comme la strucure est sauvée dans la pile de recopie et non pas dans le noeud, ce lien respecte la règle de chronologie[2] dans la pile locale.

Nous avons jusqu'à maintenant deux types de liaisons: les liaisons d'une variable à une variable plus ancienne (donc qui remonte dans la pile locale) et les liaisons des variables vers les structures (donc de pile locale vers pile de recopie). Il reste à s'assurer qu'il n'est pas possible d'avoir une liaison de la pile de recopie (donc d'une variable qui intervient dans une structure) vers la pile locale (donc pas un terme structuré).

Il est en effet possible qu'une variable X intervenant dans une strucure S soit liée à une variable B (encore dans son état libre) plus jeune que la variable A liée à la structure (voir la figure ci-dessous). Nous sommes en conflit avec la règle de chronologie. En effet, la variable A est liée à une structure dont un élément est une variable plus jeune que A.



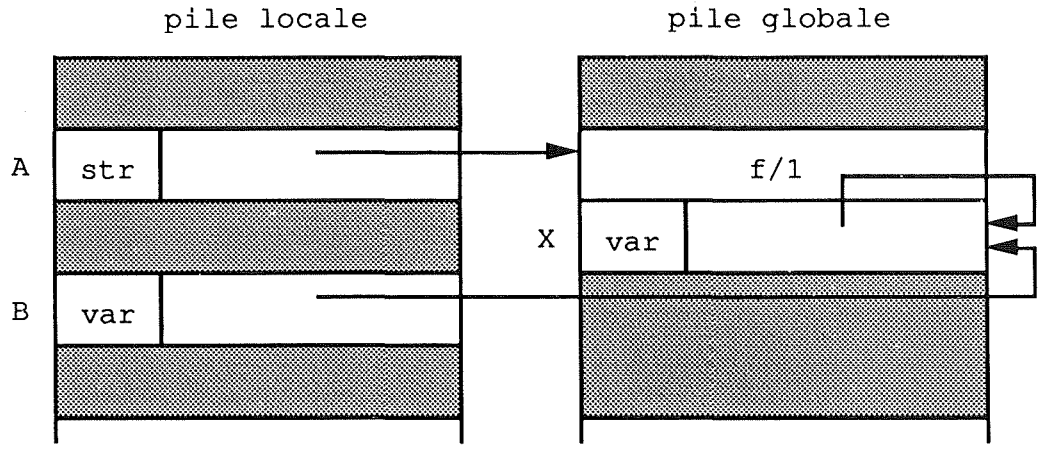
Pour éviter ce problème, lors de la création d'une structure, les liens créés à des variables intervenant dans des termes structurés ne peuvent jamais être fait de la pile de recopie (aussi appellée **pile globale**) vers la pile locale.

Exemple:

Soit une variable A liée à un terme structuré f(X) où X est unifié à B et où B appartient à un environnement plus récent que A:

---

[2] Nous nous limitons ici au cas de la représentation des structures par recopie. La technique de partage de strucure n'utilise pas de pile supplémentaire, elle doit donc mettre en oeuvre une solution adaptée.



Remarquons que le problème ne se pose que si la variable B est libre (ou liée à une variable libre, auquel cas c'est la variable libre (déréférence de B) qu'il faut lier à X.

Nous pouvons maintenant dépiler les noeuds de l'arbre ET lors de la sortie d'une procédure déterministe sans risque de perte d'information. L'algorithme d'exécution doit être modifié: en plus de chercher le but suivant à prouver (dans avancer), il faut dépiler les noeuds déterministes. La boucle "tant que le ButCourant est nil" est modifiée de la façon suivante:

- tant que le ButCourant est nil
  - si l'EnvCourant est nil, fin de l'exécution, il s'agit d'un succès
  - (**EXIT du père du ButCourant**: remonter le long de la branche qui mène au noeud courant: tous les sous-buts du père ont réussi, le père a donc aussi réussi)
  - faire pointer EnvCourant sur son père
  - faire pointer ButCourant sur le but qui suit (dans le texte du programme) le but de EnvCourant
  - dépiler le noeud du sommet de la pile si ce n'est pas un point de choix (c.à.d. si ce n'est pas le DernierPC)

2. L'appel terminal

Lorsque le dernier sous-but SB d'un but B déterministe est appelé, et que tous les autres sous-buts de la clause sont déterministes (leurs noeuds ont dès lors été dépilés suite à l'optimisation des sous-arbres déterministes), le "destin" du but B ne dépend plus que de SB. En effet, le noeud du but B ne contient plus d'informations nécessaires pour un prochain sous-but puisque SB est le dernier et ne contient plus aucune information nécessaire pour un rétro-parcours puisqu'il est déterministe ainsi que tous ses sous-buts (excepté éventuellement SB).

L'optimisation de l'appel terminal consiste donc à supprimer de la pile le noeud du but B.

Puisque SB utilise des variables de l'environement du noeud de B, si l'on supprime le noeud de B avant l'unification, nous aurons perdu des arguments du but B.

Une solution consiste à sauver les arguments (déréférencés) du but SB dans des registres[3].

Une fois cette précaution prise, le noeud B (au sommet de la pile locale) peut être supprimé de la pile locale, puisque (de par les précautions prises pour l'optimisation des sous-arbres déterministes) le respect de la règle de chronologie nous garantit qu'aucune référence n'est faite d'un autre environnement que celui du noeud de B vers des variables de l'environnement du noeud de B.

Il est néanmoins possible que certains arguments de SB, variables de l'environnement du noeud de B, fassent référence à des variables de l'environnement du noeud de B. Cela mène à la notion de **variable dangereuse** que l'on va globaliser pour éviter des références à un environnement que l'on va détruire.

Une variable X figurant comme argument du dernier sous-but d'une clause est dangereuse si la déréférence de X est une variable libre appartenant au même environnement que X.

Dès lors, si la déréférence d'une variable en position d'argument est une variable libre dans l'environnement de B, la déréférenciation de B est **globalisée**:

- une nouvelle variable libre est allouée sur la pile globale;
- l'argument et la déréférence de B sont liés à la nouvelle variable allouée sur la pile globale.

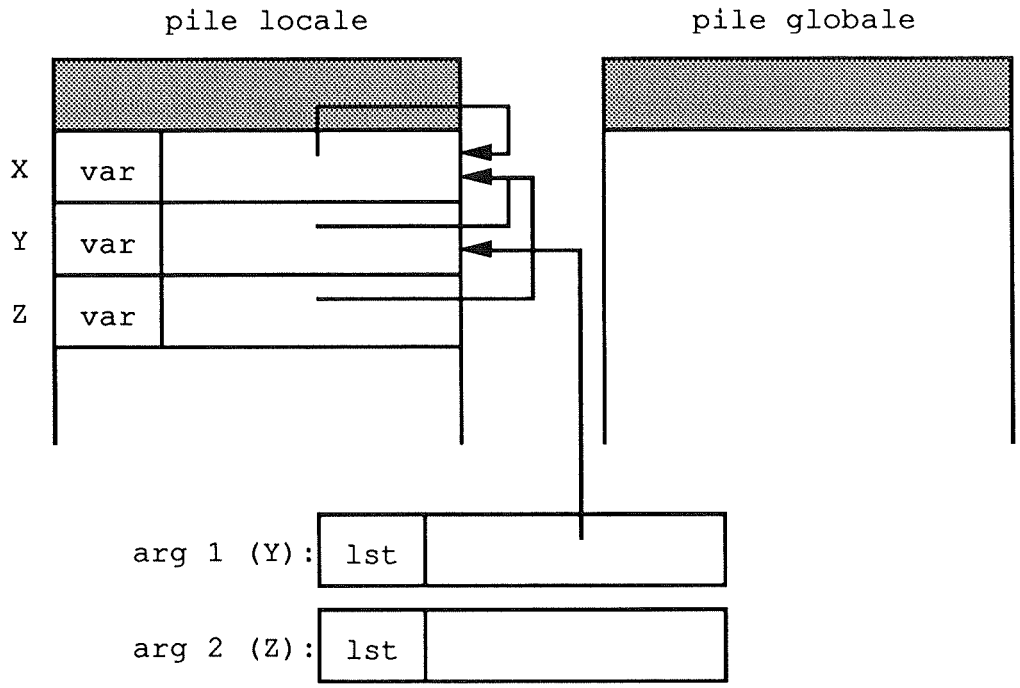
Prenons par exemple, l'environnement créé pour la clause suivante:

```
p(X) :- is(Y,X), is(Z,X), q(Y,Z)
```

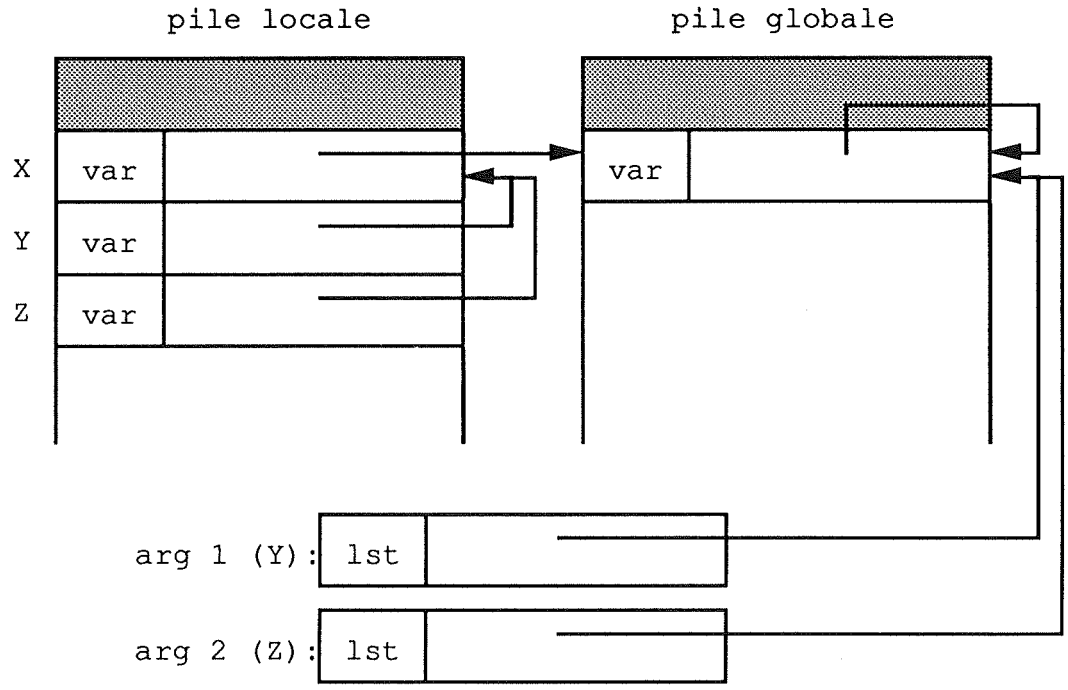
L'environnement de la clause contient les variables X, Y et Z.

Lors du chargement du premier argument on remarque que Y est une variable dangereuse (q étant en position terminale, on veut libérer l'environnement de la clause):

[3] Ce registre n'est pas forcément un registre physique, il peut bien sur s'agir d'une zône mémoire réservée à cet effet.



On va donc globaliser la variable X. Par la même occasion, la variable Z n'est plus une variable dangereuse car sa déréférence est sur la pile globale. Nous obtenons:



Nous reviendrons sur la notion de variable dangereuse dans la section suivante car cette notion doit être généralisée dans le cas de la machine abstraite (pour le trimming).

L'algorithme est modifié de façon à détecter l'appel terminal en situation déterministe avant l'unification et à dépiler le noeud dans l'affirmative:

```
PROUVER consiste à répéter:
  • sauvegarder les arguments du ButCourant
    (variables de l'EnvCourant) dans les registres
    spéciaux
  • si le but qui suit ButCourant dans le texte du
    programme est nil et que DernierPc est plus
    ancien que EnvCourant
      • faire pointer EnvCourant (au sommet de la
        pile) vers son père
      • détruire l'environnement au sommet de la
        pile (EnvCourant redevient le sommet de la
        pile)
  * (CALL du ButCourant: créer une nouvelle
    branche à partir du noeud courant, il s'agit de
    l'essai d'un but.)
  • s'il existe une clause Cl à partir de la
    ClauseCourante dans le texte du programme dont
    la tête a même foncteur et même arité que le
    ButCourant[4] et que l'unification de la tête de
    cette clause avec le ButCourant réussit (il
    faut mémoriser les liens qui sont créés lors de
    l'unification et il faut trailer les variables
    liées)
      AVANCER
  • sinon (FAIL du ButCourant)
      RETRO-PARCOURS
```

Remarquons que cette optimisation rend l'algorithme du cut incorrect. En effet, suite à l'optimisation, un noeud ne pointe plus toujours vers son noeud père. Une solution consiste à utiliser un booléen dans chaque noeud pour signaler si le noeud pointe vers son père ou non.

[4] Le but courant n'est plus envisagé avec l'EnvCourant mais avec les arguments sauvegardés.

7. La machine abstraite de Warren

La plupart des Prolog compilés sont basés sur la machine abstraite de Warren [Warren 83].

Le principe est de créer un jeu d'instructions de base comparable à une sorte de langage machine adapté à Prolog. La machine est dite abstraite dans la mesure où les instructions (abstraites) peuvent être émulées, compilées en code natif d'un processeur classique particulier ou éventuellement faire partie d'un processeur construit à cet effet (il s'agit alors d'une machine "concrète").

La machine contient une série de registres: un compteur ordinal, des registres pointeurs de pile, etc... Leurs fonctions sont pour la plupart définies par les instructions qui les utilisent.

Comme je l'ai déjà signalé plus haut, la machine abstraite utilise la méthode de recopie de structure (plutôt que la méthode de partage de structure).

Elle utilise donc trois piles: une pile locale, une pile glogale (ou de recopie) et une pile de trail. Nous supposerons que les piles sont de taille infinie.

Enfin, une zone mémoire contient le code (en instructions abstraites) des procédures du programme.

1. Les registres

La machine contient une série de registres. La plupart peuvent être mis en relation avec les variables de l'interpréteur.

`P:`	compteur ordinal, adresse de l'instruction à exécuter;
`CP:`	adresse de retour, il s'agit de la continuation;
`E:`	pointeur vers l'environnement courant (sur la pile locale);
`B:`	pointeur vers le dernier point de choix (sur la pile locale);
`TR:`	pointeur vers le sommet de la pile de trail;
`H:`	pointeur vers le sommet de la pile de recopie;
`HB:`	pointeur vers le sommet de la pile de recopie correspondant au dernier point de choix (H correspondand à B);
`S:`	pointeur de création de structure (sur la pile de recopie)[1];
`Ai:`	registres d'arguments (il y en a autant que le nombre maximum d'arguments que le Prolog concerné accepte dans un but);

[1] Ce registre est utilisé lors de la création et de l'accès aux structures sur la pile de recopie, nous ne commenterons pas davantage son fonctionnement dans la mesure où cela ne comporte pas d'intérêt particulier pour le debugger.

2. La pile locale

La pile locale enregistre les mêmes informations que dans un interpréteur mais les environnements sont indépendants des points de choix.

Les cellules de la pile peuvent contenir différents type de données:

- un pointeur vers une cellule de la pile locale ou de la pile globale;
- une adresse du code objet;
- une variable.

Les registres B et E pointent toujours vers la pile locale.Le registre E pointe vers l'environnement courant, le registre B pointe vers les informations sauvées du dernier point de choix.

Il n'y a pas de registre de sommet de pile locale. De par les optimisations (des sous-arbres déterministes et de l'appel terminal), on n'a jamais besoin d'informations plus anciennes que celle du dernier point de choix (excepté les variables qui sont globalisées le cas échéant). Dès lors, le sommet de la pile locale est déterminé par B ou E selon que B est plus haut ou plus bas que E.

3. La pile globale

La pile globale contient les structures et les variables globales, le registre `H` pointe vers son sommet.

4. La pile de trail

La pile de trail enregistre les adresses de toutes les variables qui sont liées, son sommet est pointé par le registre `TR`.

5. Le principe de base (notion intuitive)

Le principe de base consiste à compiler chaque procédure en une suite d'instructions abstraites. les registres `Ai` sont chargés avec les arguments avant l'appel de la procédure.

Un premier contrôle est fait sur l'arbre OU:

- choix de la clause;
- génération éventuelle de points de choix.

Une fois l'exécution aiguillée sur le code la bonne clause, les opérations suivantes sont accomplies:

- unifier les arguments (les registres `Ai`) avec la tête de la clause
- éventuellement création d'un environnement;
- gérer l'arbre ET de cette clause.

Gérer l'arbre ET consiste en une succession de:

- chargement des registres `Ai` avec les arguments du sous-but;

- appeler le sous-but (en mettant la continuation à jour);

- éventuellement désallouer la place des variables qui ne seront plus utilisées (il s'agit d'une généralisation de l'optimisation de l'appel terminal).

6. L'arbre OU

L'arbre OU de la machine abstraite utilise un mécanisme d'indexage (indexage à deux niveaux) des clauses sur le type de leur premier argument.

Ce mécanisme a l'avantage d'augmenter sensiblement le caractère déterministe des programmes et d'augmenter par la même occasion les optimisations des sous-arbres déterministes et de l'appel terminal.

Le principe consiste à effectuer un aiguillage de l'exécution sur les clauses dont le type du premier argument de la tête est compatible avec celui du premier argument du but (dans `A1`). Il est par exemple inutile d'envisager l'unification d'un terme fonctionnel avec une constante ou une liste.

Supposons tout d'abord qu'aucune clause de la procédure ne contienne de variable comme premier argument.

L'instruction `switch_on_term Var, Const, List, Struct` charge dans le registre `P` (compteur ordinal) l'adresse `Var, Const, List` ou `Struct` selon que la déréférence du registre `A1` est une variable, une constante, une liste ou une structure. Il s'agit d'un branchement fonction du type de la déréférence de `A1`.

Lorsque la déréférence de `A1` est une variable libre, toutes les clauses de la procédure doivent être essayées. Un arbre OU de toutes les clauses de la procédure est créé à l'aide des instructions `try_me_else, retry_me_esle` et `trust_me_else_fail`.

L'instruction `try_me_else` est utilisée pour créer un point de choix (sur la pile locale), l'instruction `retry_me_else` est utilisée pour mettre à jour le point de choix et `trust_me_else_fail` est utilisée pour détruire le point de choix avant l'exécution de la dernière clause de la procédure.

Les points de choix enregistrent les informations suivantes sur la pile globale:

- les arguments `A1` à `An` du but;

- `BCE`: pointeur vers l'environnement courant au moment de la création du point de choix;

- `BCP`: la continuation (adresse vers le code compilé du programme) au moment de la création du point de choix;

- `B'`: un pointeur vers le point de choix précédent (adresse sur la pile locale);

- `BP`: l'adresse du code (compilé) de la clause suivante (ou du groupe de clauses à cause de l'indexage);

- `TR'`: le sommet de la pile de trail au moment de la création du point de choix;

- `H'`: le pointeur vers le sommet de la pile de recopie au moment de la création du point de choix.

Ces informations permettent de gérer le rétro-parcours et sont donc utilisées lors d'un échec. Voici ce qui se passe lors d'un échec (suite à l'échec d'une unification, ou par la demande explicite de l'exécution de l'instruction abstraite fail):

- toutes les variables enregistrées dans le trail depuis TR'(B) sont rendues libres
- les arguments A1 à Ai sont chargés avec les valeurs sauvées dans le point de choix pointé par B
- les registres E, CP, TR et H sont restaurés avec les valeurs sauvées dans le dernier point de choix[2] (BCE, BCP, TR' et H')
- le registre P est chargé avec l'adresse de la clause suivante (BP)

Voici, de manière plus précise, le fonctionnement de ces trois instructions de création et de mise à jour de point de choix:

`try_me_else label`
: créer un point de choix avec `label` comme champ `BP` (adresse du code de la clause suivante), mettre à jour les registres `B` et `HB` et incrémenter le compteur ordinal (`P`). Cette instruction précède le code compilé de la première clause de l'arbre OU;

`retry_me_else label`
: mettre à jour le dernier point de choix (la clause suivante du point de choix devient `label` que l'on charge dans le champ `BP`) et passer à l'instruction suivante (incrémenter `P`). Cette instruction précède le code compilé de chaque clause de la procédure sauf la première et la dernière;

`trust_me_else_fail`
: dépiler le dernier point de choix (mettre à jour les registres `B` et `HB`) et passer à l'instruction suivante. Cette instruction précède la compilation de la dernière clause de la procédure.

Dans le cas des constantes et des structures, une fois le type de l'argument du but fixé, un second niveau d'indexation va aiguiller sur les groupes de clauses:

L'instruction `switch_on_constant(const1: label1,...,constn: labeln)` va charger dans le registre `P` l'adresse `label1`, ...,ou `labeln` selon que la valeur de la constante (déréférence de `Ai`) est `const1`, ... ou `constn`.

Tandis que l'instruction `switch_on_struct(str1: label1, ..., strn: labeln)` va charger dans le registre `P` l'adresse `label1`, ..., ou `labeln` selon que le foncteur de la structure (déréférence de `Ai`) est `str1`, ..., ou `strn`.

---

[2] Le registre `B` n'est pas mis à jour: le point de choix n'est pas dépilé.

Si, après les deux niveaux d'indexation (ou un seul pour les listes), plusieurs clauses ont le type du premier argument compatible avec la déréférence de A1, des instructions de génération de point de choix doivent être générées pour créer et mettre à jour un point de choix qui va gérer l'arbre OU de ce groupe de clauses.

Pour chacun des groupes de clauses (résultant du partitionnement des clauses par l'indexage) les instructions `try label`, `retry label` et `trust label` sont utilisées pour créer l'arbre OU du groupe. L'instruction `try` est utilisée pour créer un branchement vers la première clause du groupe et créer un point de choix; l'instruction `retry` est utilisée pour les clauses suivantes (sauf la dernière), c'est à dire mettre à jour le point de choix et passer à la clause suivante du groupe; et enfin, l'instruction `trust` est utilisée pour détruire le point de choix avant l'exécution de la dernière clause de ce groupe.

`try label` créer un point de choix (avec `P+1` comme adresse de la clause suivante (`BP`)), mettre à jour les registres `B` et `HB` et charger `label1` dans le registre `P`. Cela signifie qu'un branchement est fait vers `label` et qu'en cas d'échec ultérieur le rétro-parcours se fera à l'instruction suivante (qui sera soit un `retry` soit un `trust`);

`retry label` mettre à jour le dernier point de choix (l'adresse de la clause suivante du point de choix devient `P+1` que l'on charge dans le champ `BP`) et faire un branchement à `label`;

`trust label` dépiler le dernier point de choix, mettre à jour `B` et `HB`, et faire un branchement à `label`.

Envisageons maintenant le cas où une clause au moins de la procédure a une variable comme premier argument. Dans ce cas, il faut partitionner l'ensemble des clauses en groupes de clauses (qui respecte l'ordre de celles-ci) de manière à séparer les clauses qui ont une variable comme premier argument de celles qui ont une constante ou une structure comme premier argument.

Nous allons utiliser les instructions `try_me_else`, `retry_me_else` et `trust_me_else_fail` pour créer un arbre OU dont les branches sont les arbres OU des groupes de clauses résultants de la partition ci-dessus.

Voici un shéma de compilation qui reprend la plupart des cas de figure. J'ai supposé qu'il y avait plusieurs clauses avec $const_1$ comme premier argument (leur code se trouve à l'adresse CC_1) et une seule clause avec $const_i$ ou $const_n$ comme premier argument (dont les clauses respectives sont compilées aux adresses CV_j' et CV_k'). De même, plusieurs clauses ont str_1 comme foncteur de structure en premier argument mais une seule a la structure str_n. Enfin, remarquons que si le premier argument du but est une variable, toutes les clauses sont essayées (si nécessaire): d'abord celles qui n'ont pas une variable comme premier argument (qui sont toutes essayées car on saute en `EtVar`), ensuite les autres (qui commencent à `EtiquVar`).

J'ai également supposé que toutes les clauses ayant une variable comme premier argument ont été écrites par le programmeur à la suite de celle ayant des constantes ou des termes structurés comme premier argument.

J'ai indicé les labels de manière à montrer les liens qu'il y a entre les arbres OU. Le label CV_j, par exemple, représente l'adresse d'une des clauses (arbre ET) dont la compilation se trouve à l'adresse CV_1', CV_2',..., CV_i',... ou CV_n'.

```
try_me_else VV1
switch_on_term EtVar, EtConst, EtList, EtStruc

EtVar:
  CV1: try_me_else CV2
  CV1': arbre ET
  ...
  CVi:    retry_me_else CVi+1
  CVi': arbre ET
  ...
  CVn:    trust_me_else_fail
  CVn': arbre ET

EtConst:
  switch_on_constant( const1: CC1;
                      ...
                      consti: CVj';
                      ...
                      constn: CVk')
CC1:
  try CVa'
  ...
  retry CVb'
  ...
  trust CVc'

EtStruct:
  switch_on_struct(   str1: CS1;
                      ...
                      strn: CV1')
CS1:
  try CVd'
  ...
  retry CVe'
  ...
  trust CVf'

EtList:
  try CVx'
  ...
  retry CVy'
  ...
  trust CVz'

VV1:
  retry_me_else VV2
  arbre ET
...
VVi:
  retry_me_else VVn
  arbre ET
...
VVn:
  trust_me_else_fail
  arbre ET
```

Remarque: Suite au double indexage sur le premier argument des têtes de clauses, une même procédure peut avoir physiquement deux points de choix: un premier généré par l'instruction `try_me_else` qui gère l'arbre OU entre les procédures dont le premier argument n'est pas une variable et les autres; et un deuxième point de choix plus tard, qui gère par exemple le groupe de clauses dont le premier argument est la même constante.

7. Variables temporaires et variables permanentes.

Les **variables temporaires** sont des variables de la clause qu'il n'est pas nécessaire de sauver (dans un environnement) car elles ne servent qu'à un seul appel de l'arbre ET.

exemple:

```
p(X) :- q(X).
```

Il n'est pas nécessaire d'allouer de la place pour la variable `X` dans l'environnement de la clause. En effet, une fois l'argument de `p` (soit `X` dans le registre `A1`) fourni à la procédure `q/1` (par le registe `A1`), la variable n'est plus nécessaire à l'environnement (à condition d'avoir traité les variables dangereuses de manière adéquate bien entendu).

Il faut néanmoins s'assurer que cette variable (qui n'utilise aucun emplacement mémoire sur la pile locale ni la pile de globale) ne va pas poser de problèmes. Cela sera garanti si l'on est sûr (à la compilation) que l'unification va lier la première occurence de la variable. On en est sûr dans les cas suivants:

- la variable apparaît dans la tête de la clause: de par la règle de chronologie et les mécanismes de globalisation, cette variable sera forcément liée;
- la variable apparaît dans une structure: de par la méthode de recopie de structure, la variable aura été globalisée.
- la variable apparaît dans le dernier sous-but: de par le mécanisme de globalisation des variables dangereuses en position terminale.

Une variable temporaire est donc une variable dont la première occurence apparaît dans la tête de la clause ou dans une structure ou dans le dernier but et qui n'intervient que dans un seul but de la clause (en considérant la tête de clause comme faisant partie du premier but).

Une **variable permanente** est une variable qui n'est pas temporaire, il faut lui allouer de la place dans un environnement.

8. L'arbre ET

En plus d'implémenter l'optimisation de l'appel terminal (libération complète de l'environnement et de la continuation avant l'appel du dernier sous-but), la machine abstraite désalloue avant chaque appel d'un sous-but les variables de l'environnement qui ne sont plus nécessaires pour son exécution. Cette technique s'appelle le **trimming**.

Cela n'est bien sûr possible qu'en situation déterministe; c'est-à-dire si l'environnement courant est au sommet de la pile. Il s'agit en fait d'une généralisation de l'optimisation de l'appel terminal. Dès lors, le concept de variable dangereuse (qu'il

faudra globaliser avant sa libération de pile locale) devient plus étendu. De même que pour les variables d'un environnement libéré par l'optimisation de l'appel terminal, il faut globaliser les variables dangereuses.

De manière à pouvoir désallouer les variables au fur et à mesure, elles sont allouées dans l'environnement dans le sens inverse de leur durée de vie.

Remarquons aussi que les variables temporaires étant toujours des variables liées par nature (voir plus haut), elles ne nécessitent pas d'allocation dans l'environnement. Un environnement ne sera donc créé que s'il existe des variables permanentes dans la clause.

Nous pouvons dès lors affirmer que les faits et les clauses qui n'ont qu'un sous-but ne nécessitent pas d'environnement. Nous pouvons donc donner la forme générale de la compilation d'une clause selon les trois cas:

1. les faits:
   instructions d'unifications des `Ai` avec les arguments de la clause;
   charger `CP` dans `P` (instruction `proceed`).

2. les clauses avec un seul sous-but:
   instructions d'unifications des `Ai` avec les arguments de la clause;
   charger les arguments du sous-but dans les Ai;
   charger l'adresse du code du sous-but dans `P` (instruction `execute`);

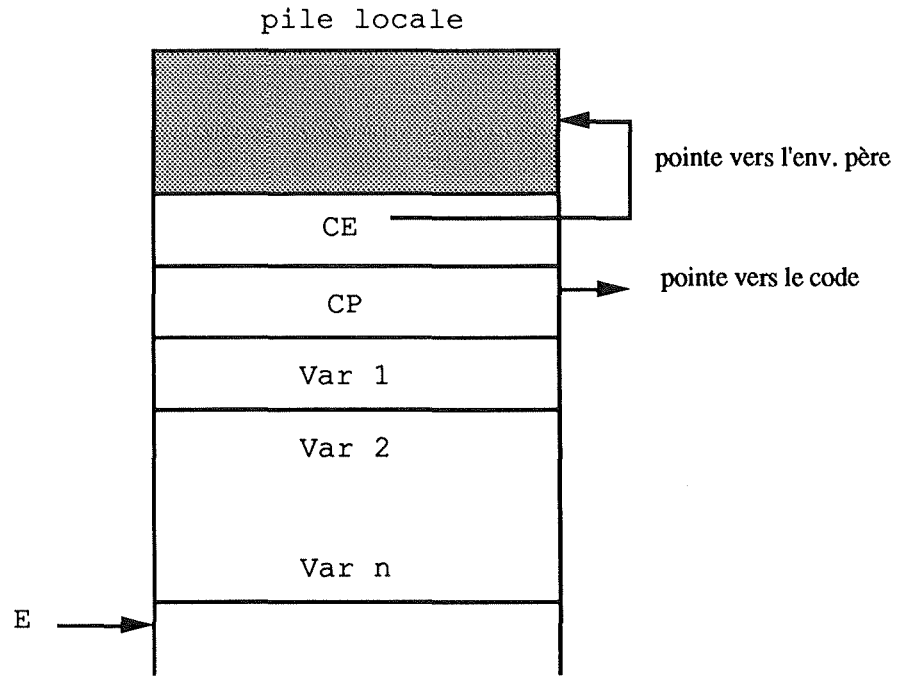
3. les clauses avec plusieurs sous-buts:
   créer un environnement sur la pile locale (instruction `allocate`)
   instructions d'unification des `Ai` avec les arguments de la clause;
   charger les arguments du premier sous-but dans les `Ai`;
   Appeler le premier sous-but et effectuer le trimming (instruction `call`);
   ...
   charger les arguments du ième sous-but dans les `Ai`;
   Appeler le ième sous-but et effectue le trimming (instruction `call`);
   ...
   charger les arguments du dernier sous-but dans les `Ai`;
   charger `CP` avec la continuation sauvée dans l'environnement et supprimer le reste de l'environnement courant (instruction `deallocate`)
   charger l'adresse du dernier sous-but dans `P` (instruction `execute`);

Remarquons que les registres CP (adresse de retour) et E (l'environnement courant) doivent être sauvés dans le cas d'une clause avec plusieurs sous-buts car ils vont être modifiés par l'instruction (ou les instructions) `call` qui suit (ou suivent).

Voici une description précise des instructions de l'arbre ET:

`allocate n`
: sauve sur la pile locale un pointeur vers l'environnement courant (`E`) et la continuation (`CP`); alloue également de la place pour un environnement de taille `n`; ce nouvel environnement devient l'environnement courant (mise à jour du registre `E`).

Typiquement, nous avons le résultat suivant sur la pile locale:



`call Proc, n`

charge la continuation (`CP`) avec l'instruction suivante (`P+1`); charge le compteur ordinal (`P`) avec l'adresse `proc` et effectue le trimming (seul un environnement de taille `n` est nécessaire pour les buts suivants; donc, si l'environnement courant est au sommet, l'allocate suivant pourra récupérer de la place).

`deallocate`

la continuation (`CP`) et l'environnement courant (`E`) sont restaurés avec les valeurs sauvées dans l'environnement courant (à désallouer). Comme la taille de la pile est déterminée par le plus grand d'entre `E` et `B`, si l'on n'est pas en situation déterministe, aucune place ne sera désallouée.

`execute proc`

le compteur ordinal (`P`) est chargé avec l'adresse `proc`.

`proceed`

charge le compteur ordinal (`P`) avec la continuation (`CP`). Il s'agit du retour du fait.

2. Spécifications fonctionnelles du debuggeur

1. Les implémentations existantes de la trace

1. Les boîtes et les portes

La plupart des implémentations existantes de debugger prolog sont basées sur les concepts de boîte et de porte.

A la section précédente, j'ai montré qu'un moment donné de l'exécution pouvait être représenté par un arbre ET de buts. Je vais montrer qu'un moment de l'exécution peut aussi être représenté par un ensemble de boîtes imbriquées.

Dans ce modèle, le passage d'un état de l'exécution à un autre n'est pas seulement représenté par des constructions et des destructions de boîtes mais aussi par un parcours au travers de celles-ci (de la même façon que l'on se déplaçait sur l'arbre ET). Le passage au travers des boîtes symbolise le passage du "**flux de l'exécution**" d'une procédure vers un de ses sous-buts. Chaque boîte représente ici une procédure avec toutes ses clauses.

Les modèles basés sur la boîte sont davantage procéduraux. L'entrée et la sortie d'une boîte peuvent être comparées à l'appel et au retour d'une sous-routine dans les langages traditionnels.

La boîte contient 4 portes, deux portes qui permettent d'entrer dans la boîte et deux portes qui permettent d'en sortir. J'en donne ici une description succinte, je décrirai deux mécanismes de construction de la structure de boîtes imbriquées ensuite. Ils seront mis en relation avec les deux manières les plus courantes de tracer l'exécution d'un programme Prolog.

- **la porte CALL** est une porte d'entrée. Lorsque l'on tente de prouver un but, une boîte est créée, et on y entre par la porte CALL.

- **la porte EXIT** est une porte de sortie; le flux de l'exécution sort de la boîte de la procédure lorsque la procédure réussit c'est-à dire lorsque tous les sous-buts d'une de ses clauses ont pu être prouvés.

- **la porte FAIL** est une porte de sortie; le flux de l'exécution sort de la boîte de la procédure par cette porte lorsque la procédure échoue.

- **la porte REDO** est une porte d'entrée; suite à l'échec d'une procédure, le flux de l'exécution entre à nouveau dans la boîte d'une procédure dont on était déja sorti par la porte EXIT afin de trouver une autre solution (retro-parcours).

2. Le modèle de la boîte pure (modèle de Byrd)

L'auteur de ce modèle a basé sa description sur une explication de la signification de chaque porte plutôt que sur une description détaillée du fonctionnement de la boîte qu'il explique à l'aide d'exemples [BYRD]. C'est le fonctionnement de la boîte que je vais décrire ici.

Une boîte doit être considérée comme un objet (dans le sens de la programmation orientée objet) qui est capable de comprendre les deux messages suivants:

- `CALL(But)` où `But` est un but prolog,

- REDO.

Il rend deux résultats:

- un nom de porte (EXIT ou FAIL),
- une liste de termes.

Le message `CALL(But)` demande à la boîte de chercher une liste de termes auquels il faut lier les variables des arguments du but `But` pour qu'il soit prouvé. Si une telle liste n'existe pas, le nom de la porte rendue en résultat est `FAIL`. Si une telle liste existe, le nom de la porte est `EXIT` et cette liste de termes est rendue en résultat.

Le message `REDO` demande, à une boîte qui a déjà reçu le message `CALL(But)`, une liste de termes auquels il faut lier les variables des arguments du but `But` pour qu'il soit prouvé mais qui est le résultat d'une autre preuve (utilise un autre agencement des clauses du programme). S'il n'est plus possible de trouver une telle liste, le nom de la porte rendue en résultat est `FAIL`. Si une telle liste existe, le nom de la porte est `EXIT` et cette liste de termes est rendue en résultat.

Le texte du programme doit être considéré comme une sorte de variable globale à laquelle on a toujours accès. Il est possible d'obtenir toutes les clauses d'une procédure donnée, il est également possible d'accéder à la première clause d'une procédure donnée et à la clause suivante d'une clause donnée d'une procédure. En ce sens, une procédure est une *liste* de clause.

Une boîte de procédure fait par contre référence à une *chaîne* (plutôt que liste) de sous-boîte de procédures dans la mesure ou elle n'accède pas seulement à la première sous-boîte ou la sous-boîte suivante d'une sous-boîte, mais aussi à la dernière sous-boîte d'une boîte ou la sous-boîte précédente d'une sous-boîte.

L'objet boîte possède les propriétés suivantes:

- `But` qui est le but que la boîte essayer de prouver,
- `ListeDeClauses` qui est une liste de clauses prolog dont les têtes ont même foncteur et même arité (elles constituent une procédure),
- `ClauseCourante` qui est une référence vers une clause de la `ListeDeClauses`,
- `ChaîneDeSousBoîtes` qui est une chaîne de boîtes (ce sont les boîtes imbriquées dans la boîte concernée),
- `ListeDeLiens` qui est une liste de liens; un lien est un couple (identificateur de variable, terme) et est une façon de représenter une subsitution.

Supposons que l'on crée une boîte et que l'on lui envoye le message `CALL(adéquat(X))` à l'aide du programme suivant:

exemple 1:

```
adéquat(X) :-
     solide(X),
     grand(X),
     long(X).

solide(Y) :-
     bas_solide(Y),
     haut_solide(Y).

grand(TOUT).

bas_solide(a).
bas_solide(b).

haut_solide(a).
haut_solide(b).

long(b).
```

La réponse sera: `EXIT` avec `X` lié à la constante `b`.

Si nous demandons ensuite à la même boîte: `REDO`, la réponse sera: `FAIL`.

On ne voit pas encore ici en quoi ce modèle peut aider à tracer le déroulement de la preuve d'un but. Cela n'est en fait possible que si l'on "entre" dans la boîte pour voir comment la preuve s'établit.

Ce qui suit est la manière d'implémenter les deux méthodes de l'objet boîte pour le modèle de la boîte pure.

Les méthodes de la boîte utilisent les variables locales suivantes:

- `Direction` est un booléen qui peut valoir `VersLAvant` ou `VersLArrière`,

- `SousBut` est une référence vers le texte d'un but,

- `SousBoîte` est une boîte,

```
La méthode CALL(LeBut) consiste à
   • sauver LeBut dans But
   • construire dans ListeDeClause une liste de
     toutes les clauses du programme dont la tête a
     le même foncteur et la même arité que le but
     But
   • ClauseCourante référence la première clause de
     ListeDeClause (nil s'il n'y en a pas)
   • ESSAYER_TOUTES_LES_CLAUSES
   • si la Direction est VersLAvant
        • fin de l'exécution, rendre EXIT et
          ListeDeLiens comme résultats.
   • sinon (la Direction est VersLArière)
        • fin de l'exécution, rendre FAIL et la
          ListeDeLiens vide comme résultat
```

De manière intuitive, `ESSAYER_TOUTES_LES_CLAUSES` consiste à essayer chacune des clauses de la procédure, en partant de la `ClauseCourante`, pour

prouver le but But. Si ces clauses ont pu prouver le but But, la direction est VersLAvant et ClauseCourante est la clause qui a servi à prouver le but, les liens qui résultent de l'exécution de la clause ont été enregistrés dans ListeDeLiens. La direction est vers l'arrière si aucune clause n'a pu servir à prouver le but.

```
ESSAYER_TOUTES_LES_CLAUSES consiste à
   • mettre la Direction VersLArrière
   • tant que la Direction est VersLArrière et que
     ClauseCourante n'est pas nil
        • si la tête de la ClauseCourante s'unifie
          avec le but But (les liens des unifications
          sont enregistrés dans ListeDeLiens)
             • faire référencer SousBut sur le premier
               sous-but de la ClauseCourante (nil s'il
               n'y en a pas)
             • mettre la Direction VersLAvant
             • UTILISER_CLAUSE(Direction, SousBut,
               SousBoîte)
        • sinon
             • rendre libre toutes les variables qui
               ont été liées (elles sont dans
               ListeDeLiens)
             • faire référencer ClauseCourante sur la
               clause suivante de ListeDeClause
```

De manière intuitive, UTILISER_CLAUSE consiste à utiliser la ClauseCourante pour prouver le but But. Donner une direction VersLAvant à UTILISER_CLAUSE lui indique que la preuve est déjà partiellement établie pour les sous-buts précédents SousBut. Donner une direction VersLArière à UTILISER_CLAUSE lui indique que SousBoîte a donné un échec et qu'il faut entamer un rétro-parcours.

```
UTILISER_CLAUSE(Direction, SousBut, SousBoîte)
  consiste à
  • tant que SousBut n'est pas nil
      • si la Direction est VersLAvant
          • créer une boîte SousBoîte
          • l'ajouter au bout de la
            ChaîneDeSousBoîtes.
          • envoyer à SousBoîte le message
            CALL(SousBut') où SousBut' vaut SousBut
            où les arguments ont été instanciés
            selon les liens de ListeDeLiens
      • sinon (la Direction est VersLArrière)
          • supprimer de ListeDeLiens, les liens qui
            sont dans la ListeDeLiens de SousBoîte
          • envoyer à SousBoîte le message REDO
      • si le résultat (du CALL ou du REDO) est
        FAIL
          • mettre la Direction VersLArrière
          • faire référencer SousBut sur le sous-but
            précédent (de SousBut) dans la
            ClauseCourante (nil si SousBut était
            déja le premier de la clause)
          • détruire SousBoîte et l'enlever de la
            ListeDesSousBoîtes
          • faire référencer SousBoîte sur la
            dernière sous-boîte de la
            ChaîneDeSousBoîte
      • sinon (le résultat est EXIT)
          • mettre la Direction VersLAvant
          • ajouter les liens rendus en résultats à
            la liste ListeDeLiens
          • faire référencer SousBut sur le but
            suivant de la ClauseCourante (nil si
            SousBut était déjà le dernier de la
            clause)
```

Le rétro-parcours consiste à faire le parcours au travers des boîtes (porte REDO) dans le sens inverse de leur création jusqu'à un point de choix et de recommencer alors l'exécution avec les mêmes instanciations que lors du CALL du point de choix mais avec une autre clause. Le point de choix qui nous intéresse est le plus jeune de ceux-ci c'est-à dire le point de choix dont la boîte a été **créée** la dernière.

```
La méthode REDO consiste à
  • faire référencer SousBoîte sur la dernière
    sous-boîte de ChaîneDeSousBoîtes (nil si la
    ChaîneDeSousBoîte est vide)
  • faire référencer SousBut sur le dernier
    sous-but de la ClauseCourante (nil s'il n'y en
    a pas)
  • mettre la Direction VersLArrière
  • UTILISER_CLAUSE(Direction, SousBut, SousBoîte)
  • si la Direction est VersLArrière
      • faire référencer ClauseCourante sur la
        clause suivante de ListeDeClause
      • ESSAYER_TOUTES_LES_CLAUSES
  • si la Direction est VersLAvant
      • fin de l'exécution, rendre EXIT et
        ListeDeLiens comme résultats.
  • sinon (la Direction est VersLArière)
      • fin de l'exécution, rendre FAIL et la
        ListeDeLiens vide comme résultat
```

Il est maintenant possible de voir le processus de preuve du but `adéquat(X)`. La trace du modèle de Byrd consiste à créer une ligne de trace chaque fois qu'un message est envoyé à une boîte et chaque fois qu'une réponse est rendue par une boîte. Afin de faciliter la lecture, chaque **invocation** (c'est-à-dire chaque boîte) reçoit un numéro. Il s'agit du numéro entre parenthèses. Un autre nombre est également affiché, il s'agit de la profondeur, c'est-à dire le nombre de boîtes imbriquées dans laquelle la boîte concernée est contenue.

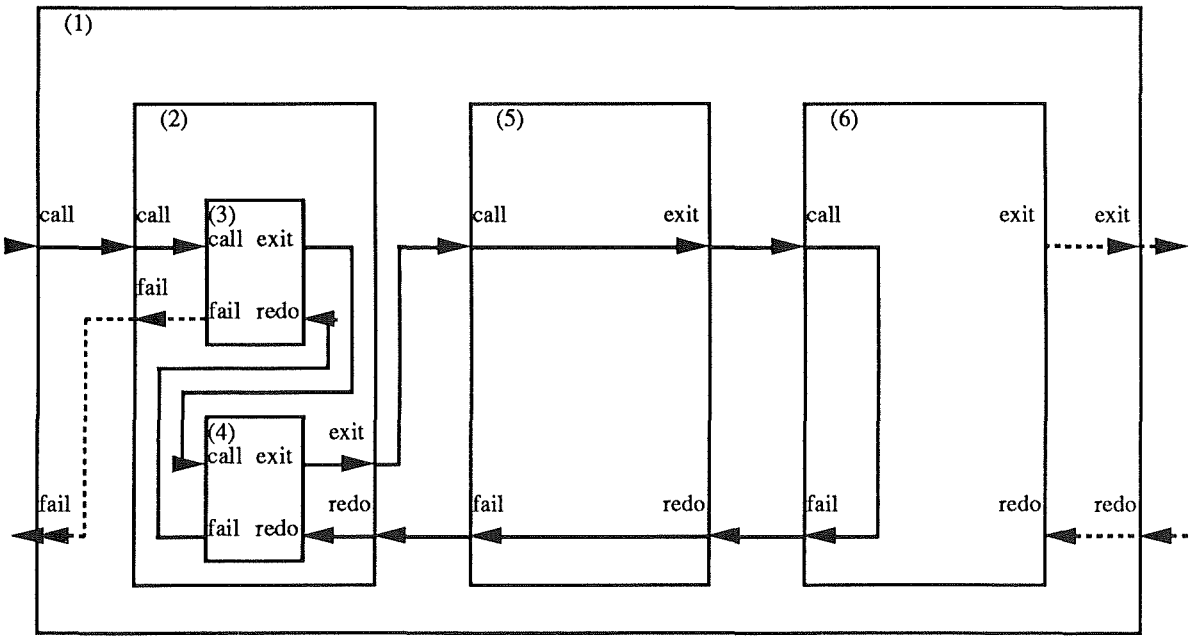
Voici ce que cela donne, suite à l'envoi du message `CALL(adéquat(X))` à une boîte:

```
(1)   0   CALL adéquat(X)
(2)   1   CALL solide(X)
(3)   2   CALL bas_solide(X)
(3)   2   EXIT bas_solide(a)
(4)   2   CALL haut_solide(a)
(4)   2   EXIT haut_solide(a)
(2)   1   EXIT solide(a)
(5)   1   CALL grand(a)
(5)   1   EXIT grand(a)
(6)   1   CALL long(a)
(6)   1   FAIL long(a)
(5)   1   REDO grand(a)
(5)   1   FAIL grand(a)
(2)   1   REDO solide(X)
(4)   2   REDO haut_solide(a)
(4)   2   FAIL haut_solide(a)
(3)   2   REDO bas_solide(X)
(3)   2   EXIT bas_solide(b)
(7)   2   CALL haut_solide(b)
(7)   2   EXIT haut_solide(b)
(2)   1   EXIT solide(b)
(8)   1   CALL grand(b)
(8)   1   EXIT grand(b)
(9)   1   CALL long(b)
(9)   1   EXIT long(b)
(1)   0   EXIT adéquat(b)
```

Lors de

```
(6)   1   FAIL long(a)
```

l'état des boîtes est le suivant:



Si nous demandons ensuite à la même boîte: REDO, nous obtenons:

```
(1)   0   REDO adéquat(X)
(9)   1   REDO long(b)
(9)   1   FAIL long(b)
(8)   1   REDO grand(b)
(8)   1   FAIL grand(b)
(2)   1   REDO solide(X)
(4)   2   REDO haut_solide(b)
(4)   2   FAIL haut_solide(b)
(3)   2   REDO bas_solide(X)
(3)   2   FAIL bas_solide(X)
(2)   1   FAIL solide(X)
(1)   0   FAIL adéquat(X)
```

Je fais dès à présent deux remarques. Tout d'abord, il n'existe à ma connaissance, sur le marché, aucune implémentation du modèle de Byrd qui trace l'exécution d'un programme en montrant dans la porte REDO les mêmes instanciations que celles montrées lors de sa porte CALL correspondante. Byrd lui-même ne donne aucune explication quant au choix qu'il fait. Il montre les instanciations qui existaient lors du passage par la dernière porte FAIL. Ce qui produit dans l'exemple 1 des instanciations insatisfaisantes du point de vue du modèle. Nous obtenons par exemple:

```
...
(5)   1   FAIL grand(a)
(2)   1   REDO solide(a)
...
(2)   1   EXIT solide(b)
...
```

Je donnerai plus loin une explication d'ordre technique de son choix.

Ensuite, j'attire l'attention sur le fait qu'un Prolog acceptable (du point de vue des performances) ne pourrait pas être implémenté selon ce modèle. En effet, contrairement à l'implémentation minimum que j'ai décrite à la section précédente, il n'est pas possible d'implémenter ce modèle sous forme de pile (où chaque élément de la pile est une boîte). La raison en est que les boîtes (contrairement aux noeuds de l'arbre) n'ont pas une taille fixe. La chaîne de sous-boîtes d'une boîte donnée évolue tout au long de l'exécution. La différence fondamentale entre les deux modèles est que, dans celui-ci, les boîtes contiennent des références vers leurs sous-boîtes. Cette information est inutile du point de vue strict de l'exécution.

Il est bien entendu possible de se passer de cette chaîne de références. Pour trouver le dernier noeud fils d'un noeud donné, il faut remonter la pile jusqu'à un noeud qui a ce noeud donné comme noeud père. Cela peut s'avérer très coûteux lors du rétro-parcours. Je reviendrai plus loin sur une manière moins couteuse de résoudre le problème tout en gardant une pile dont les éléments ont une taille fixe.

3. Trace orientée implémentation

Entre autres pour les raisons citées ci-dessus, certaines implémentations de Prolog ont préféré une trace plus proche de l'implémentation que celle du modèle de Byrd. C'est la raison pour laquelle je l'appellerai la trace orientée implémentation.

Voyons tout d'abord la trace qui résulterait de l'exécution de l'exemple 1 sur l'implémentation mimimum décrite plus haut avec la suite de but Bs égale au but `adéquat(X).`

Considérons les noeuds de l'arbre comme des boîtes. Les noeuds fils d'un noeud père deviennent les sous-boîtes d'une boîte. La porte CALL de la boîte correspond à la

création du noeud et de la branche qui y mène, la porte FAIL correspond à l'échec d'une unification, la porte REDO correspond à la destruction de tous les noeuds qui ont été créés depuis le dernier point de choix et à la réexécution de ce point de choix et enfin la porte EXIT correspond au déplacement du noeud courant vers son noeud père suite à la réussite du dernier sous-but d'une clause (voire l'algorithme minimum d'exécution).

```
(1)  0  CALL adéquat(X)
(2)  1  CALL solide(X)
(3)  2  CALL bas_solide(X)
(3)  2  EXIT bas_solide(a)
(4)  2  CALL haut_solide(a)
(4)  2  EXIT haut_solide(a)
(2)  1  EXIT solide(a)
(5)  1  CALL grand(a)
(5)  1  EXIT grand(a)
(6)  1  CALL long(a)
(6)  1  FAIL long(a)
     (récupération de l'état d'exécution du dernier point de choix)
(4)  2  REDO haut_solide(a)
(4)  2  FAIL haut_solide(a)
     (récupération de l'état d'exécution du dernier point de choix)
(3)  2  REDO bas_solide(X)
(3)  2  EXIT bas_solide(b)
(7)  2  CALL haut_solide(b)
(7)  2  EXIT haut_solide(b)
(2)  1  EXIT solide(b)
(8)  1  CALL grand(b)
(8)  1  EXIT grand(b)
(9)  1  CALL long(b)
(9)  1  EXIT long(b)
(1)  0  EXIT adéquat(b)
```

La différence se situe en fait uniquement lors du rétro-parcours. Tout d'abord, la porte FAIL est traversée à chaque échec de clause et non pas uniquement lors de l'échec de la procédure tout entière. Ensuite, plutôt que de parcourir toutes les boîtes en sens inverse, on reprend immédiatement après le FAIL l'état d'exécution du dernier point de choix.

Cela entraîne plusieurs conséquences:

- Comme on peut le constater dans la trace ci-dessus, les rétro-parcours au travers des sous-arbres déterministes ne sont pas montrés. Ou, en d'autres termes, on n'entre pas par la porte REDO (et donc forcément pas de FAIL) dans une boîte qui ne contient plus de point de choix (pas de REDO de `grand(a)`).

- On ne montre pas non plus le passage au travers de la porte REDO d'une boîte qui n'est pas le dernier point de choix (pas de REDO de `solide(X)`). Cela signifie que contrairement au modèle de Byrd, toutes les portes de sortie ne correspondent pas à une porte d'entrée.
  Notons que cela n'est pas dû au fait que les points de choix sont chaînés. En effet, même si lors du rétro-parcours, la pile était dépilée environnement par environnement jusqu'à celui d'un point de choix, on rencontrerait toujours l'environnement d'une sous-boîte avant celui de sa boîte mère. Pour être capable de montrer la porte REDO d'une boîte avant

celles de ses sous-boîtes, il nous faut, comme je l'ai déjà dit plus haut, un mécanisme coûteux inutile du point de vue strict de l'implémentation.

- On montre un FAIL dès l'échec d'une clause même si la procédure (la boîte) n'a pas encore échouée. Cela signifie, par exemple, que lorsqu'un but ne peut s'unifier qu'avec la troisième clause d'une procédure, la trace est constituée d'un CALL et d'une succession de deux FAIL et REDO.

- Certaines boîtes peuvent ne pas avoir eu de porte de sortie (FAIL). On ne montre en effet que le FAIL du but dont l'unification a échoué mais pas les FAIL des boîtes mères qui échouent du fait de l'échec de ses sous-boîtes.

En fait, le rétro-parcours est ici un retour direct (REDO) à l'état de l'exécution le plus récent (le dernier point de choix) suite à un échec (FAIL).

Dans la pratique, les concepteurs (qui choisissent de ne pas implémenter le modèle de la boîte pure), corrigent néanmoins les deux derniers points. Suite à un FAIL, une séquence de porte FAIL est traversée: toutes celles des boîtes des procédures qui sont plus récentes que le dernier point de choix de manière à sortir de toutes les boîtes qui ont échouées avant de réentrer dans le point de choix. Une porte FAIL et la porte REDO qui la succède ne sont pas tracées si l'échec n'était pas celui de la dernière clause de la procédure.

4. Critiques

1. Trop d'informations au rétro-parcours dans le modèle de Byrd

Le modèle de la boîte pure (modèle de Byrd) , montre le cheminement complet au travers de toutes les boîtes, y compris le cheminement à l'intérieur d'une boîte et de toutes ses sous-boîtes alors que l'on sait à priori qu'aucun point de choix ne se trouve dans celle-ci. C'est le cas par exemple du REDO de la boîte (5) de l'exemple 1. Du point de vue du modèle de l'arbre, cela correspond à parcourir tout un sous-arbre qui ne contient pas de point de choix. L'inconvénient est que cette information est très peu souvent utile et peut, dans des programmes fortement déterministes, encombrer la trace d'une quantité élevée de FAIL et de REDO.

Dans la suite, je qualifierai ces portes de **REDO déterministe** et **FAIL déterministe**.

2. Pas assez d'informations au rétro-parcours dans le modèle orienté implémentation

Le modèle orienté implémentation, quant à lui, ne montre que l'ensemble des procédures qui échouent et la procédure qui est appelée à nouveau (le point de choix). En d'autres termes, il sort de une ou plusieurs boîtes mais ne montre pas les boîtes dans lesquelles il entre pour atteindre le point de choix. Du point de vue du modèle de l'arbre cela signifie que l'on montre une partie du rétro-parcours (les FAIL qui remontent dans l'arbre) et pas la descente vers le point de choix.

Lors du debugging, il peut dans certains cas être difficile de suivre le cheminement du rétro-parcours.

3. Le rétro-parcours au niveau des clauses

Le modèle de la boîte ne montre pas le rétro-parcours d'une procédure dont on n'est pas encore sorti.

Exemple 2:

```
convient(bien, X) :-
     grand(X),
     long(X).
convient(moins_bien, X) :-
     long(X).

long(a).
```

si l'on demande `CALL convient(Comment, Quoi):`

```
(1)  0  CALL convient(Comment, Quoi)
(2)  1  CALL grand(X)
(2)  1  FAIL grand(X)
(3)  1  CALL long(X)
(3)  1  EXIT long(a)
(1)  0  EXIT convient(moins_bien, a)
```

Le but `convient(Comment, Quoi)` n'a pas encore échoué lorsque `grand(X)` échoue car une autre clause doit encore être essayée. On ne sort donc pas de la boîte numéro (1) par la porte `FAIL`. Dans ces conditions, il n'est pas possible (dans le cadre de ce modèle) de montrer le rétro-parcours par une porte `REDO` de la boîte numéro (1). En effet, on ne peut entrer dans une boîte (de même invocation) dont on n'est pas sorti. Ce rétro-parcours se passe au niveau des clauses et non au niveau de la procédure. Comme nous l'avons vu à la section précédente, ce rétro-parcours existe néanmoins au niveau de l'implémentation.

Dans bien des cas, ce rétro-parcours est une information importante à l'utilisateur. Il n'est pas toujours aussi évident (à l'instar de cet exemple-ci) que `CALL long(X)` appartient à la deuxième clause de `convient/2` et non à la première. Il est nécessaire pour l'utilisateur de savoir quelle clause de quelle boîte est exécutée.

2. Recherche d'un modèle de trace plus précis

1. Etendre le modèle de la boîte

Notre but est de supprimer les 3 critiques faites au sujet des deux modèles décrits ci-dessus, à savoir:

- ne pas montrer le rétro-parcours au travers des procédures (devenues) déterministes. Autrement dit, ne pas entrer dans les boîtes des procédures qui n'ont plus de clauses à essayer.

- montrer néanmoins le rétro-parcours au travers des boîtes qui sont ou contiennent des points de choix de manière à avoir pour toute sortie de boîte une et une seule entrée de boîte qui y correspond et de cette façon pouvoir suivre complètement le cheminement de l'exécution d'un but qui a échoué jusqu'au point de choix le plus récent (le chemin le plus court dans l'arbre ET).

- montrer le rétro-parcours à l'intérieur de la boîte d'une procédure, c'est-à-dire le rétro-parcours au niveau des clauses.

Les deux premiers points peuvent se résoudre facilement, il suffit de partir du modèle de la boîte pure mais de le modifier de façon à ne pas montrer les `REDO` (et les `FAIL` qui y correspondent) des boîtes qui ne sont plus des point de choix et qui n'ont plus de sous-boîtes qui sont des points de choix. Dans la suite, nous qualifierons ces portes de déterministes.

En ce qui concerne le rétro-parcours au niveau des clauses d'une boîte, nous allons "ouvrir" la boîte de la procédure et préciser son fonctionnement (ses méthodes) en utilisant un nouveau type de boîte: la boîte d'une clause (par opposition à la boîte d'une procédure).

Dans ce modèle, la boîte de la procédure ne contient plus une liste de procédures mais plutôt une liste de boîtes de clause.

De même que la boîte de la procédure, la boîte de la clause comporte 4 portes:

- la porte `UNIFY` est une porte d'entrée, le flux de l'exécution entre par cette porte dans la boîte de la clause lorsque l'on tente une unification de cette clause afin de l'utiliser pour la preuve d'un but.

- la porte `EXIT_C` est une porte de sortie, le flux de l'exécution sort de la boîte de la clause par cette porte lorsque celle-ci a réussi.

- la porte `FAIL_C` est une porte de sortie, le flux de l'exécution sort de la boîte de la clause par cette porte lorsque la clause a échoué. Cette clause ne peut servir à prouver le but donné.

- la porte `RE-ENTER` est une porte d'entrée, le flux de l'exécution entre dans la boîte de la clause par cette porte afin de trouver une autre solution avec la même clause.

En considérant la boîte de la clause comme un objet, on peut également lui envoyer deux messages: `UNIFY(But)` ou `RE-ENTER`.

Le message `UNIFY(But)` demande à la boîte de chercher une liste de termes auquels il faut lier les variables des arguments du but `But` pour qu'il soit prouvé. Si une telle liste n'existe pas, le nom de la porte rendue en résultat est `FAIL_C`. Si une telle liste existe, le nom de la porte est `EXIT_C` et cette liste de termes est rendue en résultat. La boîte de la clause ne doit utiliser que la clause `Clause` pour effectuer cette preuve (mais ses sous-boîtes peuvent ulitiser toutes les clauses du programme).

Le message `RE-ENTER(But)` demande, à une boîte qui a déjà reçu le message `UNIFY(But)`, une liste de termes auquels il faut lier les variables des arguments du but `But` pour qu'il soit prouvé mais qui est le résultat d'une autre preuve (utilise un autre agencement des clauses du programme). S'il n'est plus possible de trouver une telle liste, le nom de la porte rendue en résultat est `FAIL_C`. Si une telle liste existe, le nom de la porte est `EXIT_C` et cette liste de termes est rendue en résultat. La boîte de la clause ne doit utiliser que la clause `Clause` pour effectuer cette preuve.

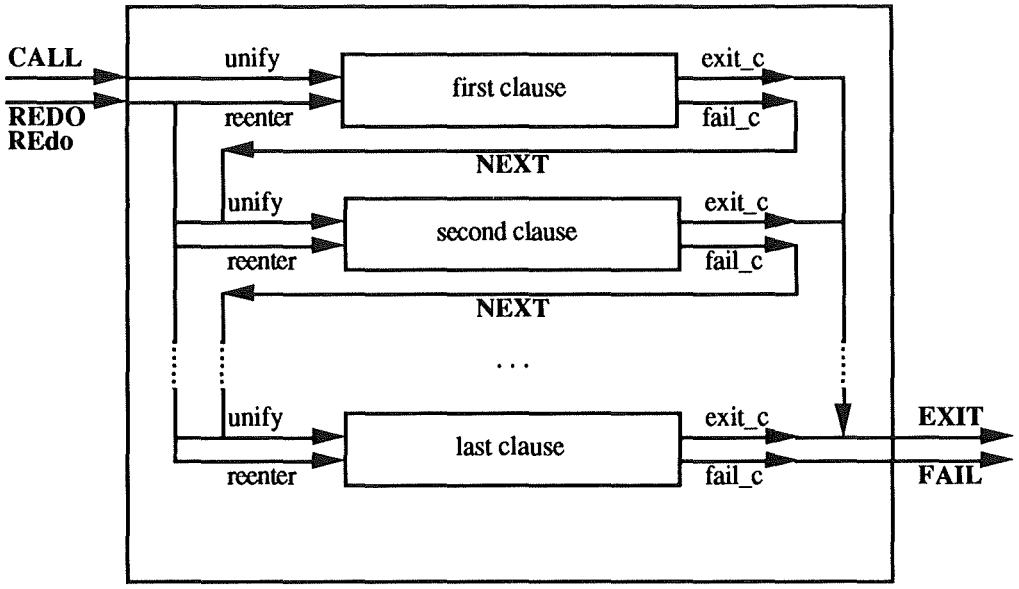
L'objet boîte de procédure possède les propriétés suivantes:

- `But` qui est le but que la boîte essayer de prouver,
- `ListeDeBoîtesDeClause` qui est une liste de boîtes de clauses Prolog dont les têtes ont même foncteur et même arité (elle constitue une procédure),
- `ClauseCourante` qui est une référence vers une boîte de clause de la `ListeDeBoîtesDeClause`.

L'objet boîte de clause possède les propriétés suivantes:

- `But` qui est le but que la boîte doit essayer de prouver,
- `Clause` qui est une clause Prolog.
- `ChaîneDeSousBoîtes` qui est une chaîne de boîtes (ce sont les boîtes de procédure imbriquées dans la boîte de clause concernée),
- `ListeDeLiens` est une liste de liens.

Les boîtes de clause s'agencent de la manière suivante dans la boîte de la procédure:



Les méthodes qui suivent utilisent une variable `Direction` dont le but est de savoir si l'on est en position d'échec ou de succès (donc avance ou rétro-parcours). La direction est vers l'avant tant que les sous-boîtes réussissent, elle passe vers l'arrière dès qu'une sous-boîte échoue et le reste jusqu'à ce qu'une sous-boîte puisse fournir une autre solution, auquel cas la direction repasse vers l'avant.

Les méthodes `CALL` et `REDO` de la boîte de la procédure deviennent celles-ci:

```
La méthode CALL(LeBut) consiste à
    • sauver LeBut dans But
    • construire dans ListeDeBoîtesDeClause une liste
      de boîtes de clause contenant chacune une
      clause du programme dont la tête a le même
      foncteur et la même arité que le but But
    • faire référencer ClauseCourante sur la première
      boîte de ListeDeBoîtesDeClause (nil s'il n'y en
      a pas)
    • ESSAYER_TOUTES_LES_CLAUSES
    • si la Direction est VersLAvant
        | • fin de l'exécution, rendre EXIT et la liste
        |   de liens de la boîte de la ClauseCourante.
    • sinon (la Direction est VersLArrière)
        | • fin de l'exécution, rendre FAIL et une
        |   liste de liens vide comme résultat
```

```
ESSAYER_TOUTES_LES_CLAUSES consiste à
  • mettre la Direction VersLArrière
  • tant que la Direction est VersLArrière et que
    ClauseCourante n'est pas nil
      • envoyer le message UNIFY(But) à la boîte
        ClauseCourante
      • si le résultat est FAIL_C (direction est
        VersLArrière)
          • faire référencer ClauseCourante sur la
            boîte suivante de ListeDeBoîtesDeClause
      • sinon (le résultat est EXIT_C et direction
        est VersLAvant)
          • la liste de liens à rendre en résultat
            est la ListeDeLiens de la ClauseCourante
```

```
La méthode REDO(But) consiste à
  • envoyer le message RE-ENTER à la boîte
    ClauseCourante
  • si le résultat est FAIL_C
      • faire référencer ClauseCourante sur la
        boîte suivante de ListeDeBoîtesDeClause
      • ESSAYER_TOUTES_LES_CLAUSES
  • si la Direction est VersLAvant
      • fin de l'exécution, rendre EXIT et la
        ListeDeLiens de la ClauseCourante comme
        résultat.
  • sinon (la Direction est VersLArière)
      • fin de l'exécution, rendre FAIL et une
        liste de liens vide comme résultat
```

Les méthodes de la boîte de clause sont les suivantes:

```
UNIFY(LeBut)
  • sauver LeBut dans But
  • si la tête de la Clause s'unifie avec le but
    But (les liens des unifications sont
    enregistrés dans ListeDeLiens)
      • faire référencer SousBut sur le premier
        sous-but de la Clause (nil s'il n'y en a
        pas)
      • mettre la Direction VersLAvant
      • UTILISER_CLAUSE(Direction, SousBut,
        SousBoîte)
  • sinon
      • mettre une liste vide dans ListeDeLiens
```

```
RE-ENTER
   | • faire référencer SousBoîte sur la dernière
   |   sous-boîte de ChaîneDeSousBoîtes (nil si la
   |   ChaîneDeSousBoîte est vide)
   | • faire référencer SousBut sur le dernier sous-
   |   but de la Clause (nil s'il n'y en a pas)
   | • mettre la Direction VersLArrière
   | • UTILISER_CLAUSE(Direction,SousBut,SousBoîte)
```

`UTILISER_CLAUSE(Direction,SousBut,SousBoîte)` reste identique excepté que l'on travaille sur la `Clause` de la boîte plutôt que sur la `ClauseCourante`.

Voyons ce que devient la trace de l'exemple 2 dans ce modèle si l'on montre les passages par les portes des boîtes de procédures et des boîtes de clauses:

```
(1)   0   CALL convient(Comment, Quoi)
          UNIFY convient(Comment,Quoi)
(2)   1   CALL grand(X)
          UNIFY grand(X)
          FAIL_C grand(X)
(2)   1   FAIL grand(X)
          FAIL_C convient(Comment,Quoi)
          UNIFY convient(Comment,Quoi)
(3)   1   CALL long(X)
          UNIFY long(X)
          EXIT_C long(a)
(3)   1   EXIT long(a)
          EXIT_C convient(moins_bien, a)
(1)   0   EXIT convient(moins_bien, a)
```

L'intérêt de cette trace est qu'il est maintenant possible de suivre le rétro-parcours de la première clause de `convient/2` vers la deuxième grâce aux deux lignes:

```
FAIL_C convient(Comment,Quoi)
UNIFY convient(Comment,Quoi)
```

Le désavantage quant à lui est évident, la trace est encombrée d'informations. Nous allons voir à la section suivante les informations que l'on peut supprimer et celles qui sont pertinentes.

2. Nettoyage des informations superflues

En regardant le graphique de la boîte, nous pouvons faire les conclusions suivantes:

- une porte `CALL` est toujours suivie d'une porte `UNIFY`. Or l'état du système n'a pas changé entre le passage du flux par la porte `CALL` et le passage par la porte `UNIFY`. La porte `UNIFY` n'apporte donc aucune information supplémentaire lorsqu'elle suit la porte `CALL`.

- une porte `EXIT` est toujours précédée d'une porte `EXIT_C`. De même que pour les portes `CALL` et `UNIFY`, l'état de l'exécution n'a pas changé, la porte `EXIT_C` n'apporte aucune information supplémentaire.

- une porte FAIL est toujours précédée d'une porte FAIL_C. La porte FAIL_C est donc toujours redondante lorsqu'elle précède la porte FAIL.

Si nous supprimons de la trace les portes EXIT_C, FAIL_C et UNIFY dans les conditions décrites ci-dessus, les portes des boîtes de clauses n'apparaissent plus dans la trace que dans les séquences suivantes:

- REDO suivit de RE-ENTER, il s'agit du rétro-parcours qui entre à nouveau dans une boîte de clause parce qu'il y a une sous-boîte d'un point de choix à l'intérieur de cette boîte (cette clause n'est pas véritablement ré-exécutée).

- REDO suivit de UNIFY, il s'agit de l'aboutissement du rétro-parcours, la clause est exécutée car c'est la clause suivante à essayer du dernier point de choix.

- FAIL_C suivit de UNIFY, il s'agit du rétro-parcours d'une clause vers la suivante à l'intérieur de la boîte de la procédure qui est le dernier point de choix (sans passer par une des portes de sa boîte de procédure). Il s'agit du rétro-parcours regretté dans le modèle de la boîte pure.

3. Trace résultante

La trace résultante consiste donc à donner un nom à chacune des 3 séquences ci-dessus afin de différencier le REDO qui entre dans la boîte de la procédure qui est le point de choix le plus récent (REDO -> UNIFY) du REDO qui entre dans la boîte d'une procédure qui contient la boîte du point de choix le plus récent (REDO -> RE-ENTER) et enfin du rétro-parcours qui passe d'une clause à l'autre de la procédure (FAIL_C -> UNIFY).

Les deux premières sont des portes de la procédure, nous les noterons respectivement REDO et REdo. La troisième est le compactage de deux portes de clauses (FAIL_C et UNIFY) et sera notée NEXT.

L'exemple 1 devient:

```
(1)  0  CALL adéquat(X)
(2)  1  CALL solide(X)
(3)  2  CALL bas_solide(X)
(3)  2  EXIT bas_solide(a)
(4)  2  CALL haut_solide(a)
(4)  2  EXIT haut_solide(a)
(2)  1  EXIT solide(a)
(5)  1  CALL grand(a)
(5)  1  EXIT grand(a)
(3)  1  CALL long(a)
(3)  1  FAIL long(a)
(2)  1  REdo  solide(a)
(4)  2  REDO  haut_solide(a)
(4)  2  FAIL haut_solide(a)
(3)  2  REDO bas_solide(X)
(3)  2  EXIT bas_solide(b)
(5)  2  CALL haut_solide(b)
(5)  2  EXIT haut_solide(b)
(2)  1  EXIT solide(b)
(6)  1  CALL grand(b)
(6)  1  EXIT grand(b)
(7)  1  CALL long(b)
(7)  1  EXIT long(b)
(1)  0  EXIT adéquat(b)
```

et l'exemple 2:

```
(1)  0  CALL convient(Comment, Quoi)
(2)  1  CALL grand(X)
(2)  1  FAIL grand(X)
(1)  0  NEXT  convient(Comment,  Quoi)
(3)  1  CALL long(X)
(3)  1  EXIT long(a)
(1)  0  EXIT convient(moins_bien, a)
```

4. Affichage des instantiations dans une porte

Lors du passage par une porte d'une boîte, le debugger ne montre pas le but tel qu'il l'a été écrit dans le texte du programme. En effet, l'utilisateur est d'avantage intéressé par les instances des variables.

Dès lors, lorsqu'une variable est instanciée à une constante, la ligne de trace est montrée avec les subsitutions qui résultent des unifications, donc avec cette constante à la place de la variable. Ce procédé semble naturel lorsque la variable est complettement instanciée.

Le problème est néanmoins plus subtil pour les variables encore libres. En effet, le nom que l'utilisateur a donné à cette variable n'est pas toujours accessible lors de l'exécution. Différents choix peuvent être pris concernant le nom à donner à une variable libre dans la trace d'un but.

La plupart des debuggers tracent les variables libres d'un but en imprimant sa position dans les piles Prolog plutôt que le nom donné aux variables par l'utilisateur dans le code source. La problématique est aussi bien d'ordre technique que d'ordre fonctionnel. Je vais énoncer ici les différents éléments qui dirigent les choix des implémenteurs ainsi que des alternatives.

1. Affichage orienté implémentation

Beaucoup de debuggers spécifient les variables par leur adresse sur les piles Prolog.

D'un point de vue fonctionnel, on peut avancer les arguments suivant concernant cette méthode:

- Un nom tel que `Liste` est plus expressif que `g23845`.

- Ces adresses ont le désavantage de devenir rapidement élevées et donc d'être peu lisibles et difficilement mémorisables.

+ Ces noms ont l'avantage d'identifier des variables de manière univoque. Les programmeurs utilisent souvent des mêmes noms de variables pour des procédures différentes (éventuellement écrites par des personnes différentes). Dans les programmes récursifs un même nom de variable correspond à plusieurs instances différentes.

D'un point de vue technique, l'argument est direct:

+ Son implémentation est de la plus grande simplicité et ne coûte ni place mémoire ni temps d'exécution car elle ne nécessite aucune gestion particulière pour le debugger.

2. Affichage orienté code source

La solution qui consite à spécifier les variables par le nom que le programmeur leur a donné dans le texte source du programme comporte en fait les avantages qui faisaient les inconvénients de la méthode orientée implémentation et vice versa:

+ Un nom de variable est plus expressif qu'une adresse.

+ Un nom de variable est plus lisible et plus mémorisable qu'une adresse.

- Un nom de variable n'identifie pas une variable univoquement.

Les conséquences sur l'implémentation nécessitent davantage d'explications et dépendent des choix faits sur le nom à tracer.

1. Quel nom du texte source utiliser

Suite aux unifications, une variable intervenant dans un but peut concerner plusieures variables. C'est le cas de la variable que l'utilisateur a appelé `X` dans la clause:

```
unifie(X,X).
```

lors du passage par la porte `EXIT` de la boîte dont le but est `unifie(Y,Z)`.

En effet, dans cette porte, les variables X, Y et Z partagent exactement le même destin, il s'agit en fait de la même instance.

Il y a principalement deux choix possibles en ce qui concerne le nom du texte source à utiliser dans la trace d'un programme: le nom que le programmeur a donné à la variable dans la clause où elle a été créée[1] ou le nom de la variable dans la clause dont on trace une porte (la clause courante).

Dans le premier cas, un tel nom existe toujours mais pas dans le second. Analysons cela en considérant trois type de variables.

a) une variable libre à tracer dans une porte de la clause dans laquelle elle a été créée:

Ce cas est le cas le plus simple, le nom que l'utilisateur a donné à la variable dans la clause dans laquelle elle a été créée est le même que le nom qu'il lui a donné dans la clause dont on passe une porte.

Du point de vue de l'implémentation, il n'y a pas de problèmes majeurs, il suffit de chercher le nom de la variable dans la clause courante.

b) une variable libre, n'intervenant pas dans une structure, à tracer dans une porte d'une autre clause que celle dans laquelle elle a été créée:

Remarquons que dans ce cas, la variable est forcément une variable en position d'argument dans la clause. D'une manière imagée, on peut dire qu'elle est "reçue de plus haut" dans l'arbre d'exécution.

Il y a ici moyen de donner deux noms différents à la variable.

Exemple:

[1] En terme d'implémentation, il s'agit du nom de la déréférence de la variable; en terme fonctionnel, il s'agit du nom de la variable la plus ancienne parmis toutes les variables instanciées à cette variable.

```
p :-
     q(X,X).

q(A,B) :-
     vaut_a(A).

vaut_a(a).
```

Si l'on décide de donner à une variable le nom que l'utilisateur lui a donné dans la clause dont on trace la porte, nous obtenons la trace suivante pour la preuve du but `p`:

```
(1)  0  CALL p
(2)  1  CALL q(X,X)
(3)  2  CALL vaut_a(A)
(3)  2  EXIT vaut_a(a)
(2)  1  EXIT q(a,a)
(1)  0  EXIT p
```

La trace ne montre pas dans la porte `CALL` de `vaut_a` que la variable `A` est en fait la variable `X`. Il n'est pas toujours aussi simple de voir cette relation (qui est par contre montrée si l'on donne l'adresse de la variable).

Tandis que si l'on décide de montrer le nom que le programmeur a donné à la variable dans la clause dans laquelle elle a été créée:

```
(1)  0  CALL p
(2)  1  CALL q(X,X)
(3)  2  CALL vaut_a(X)
(3)  2  EXIT vaut_a(a)
(2)  1  EXIT q(a,a)
(1)  0  EXIT p
```

la raison de `q(a,a)` comme résultat est beaucoup plus claire.

Néamoins, dans d'autres cas, l'utilisateur peut au contraire préférer voir le nom qu'il avait donné à la variable dans la clause concernée. Il cherche en fait la sémentique de la variable dans le contexte de la boîte dont on trace une porte et non pas l'hystorique de la variable.

c) une variable libre, intervenant dans une structure, à tracer dans une porte d'une autre clause que celle dans laquelle elle a été créée:

Dans ce cas, la variable peut aussi bien être en position d'argument (venir d'en haut), créée dans la clause elle-même ou être créée lors de l'exécution d'un sous-but de la clause auquel cas elle est en position de résultat (de manière imagée, elle vient du bas de l'arbre).

Comme je l'ai dit plus haut, nous supposons que l'on veut montrer l'instanciation maximum d'une variable (si une variable X est une liste de deux variables libres, nous voulons le montrer). Dans de telles conditions,

Il peut être ici impossible de montrer le nom que l'utilisateur a donné à la variable dans la clause dont on trace la porte dans la mesure où elle n'a pas de nom dans cette clause, elle fait en fait partie d'un terme structuré. Voici un exemple:

Exemple:

```
p :-
     q(X,X).

q(A,B) :-
     vaut_singleton(A).

vaut_singleton([E]).
```

essayons de donner à une variable le nom que l'utilisateur lui a donné dans la clause dont on trace la porte. Nous obtenons la trace suivante pour la preuve du but `p`:

```
(1)  0  CALL p
(2)  1  CALL q(X,X)
(3)  2  CALL vaut_singleton(A)
(3)  2  EXIT vaut_singleton([E])
(2)  1  EXIT q([?],[?])
(1)  0  EXIT p
```

Dans la mesure où la variable `A` a été instaciée à la liste `[E]`, nous voudrions le tracer mais cette information n'est pas disponible dans la clause courante, il faut alors donner le nom que l'utilisateur a donné à la variable dans la clause où elle a été créée. Dans ce cas-ci, elle "vient d'en bas".

Si l'on veut éviter de devoir donner le nom que l'utilisateur a donné à la variable dans la clause où elle a été créée (qui est beaucoup plus coûteux du point de vue de l'implémentation), nous devrons donner un nom conventionnel à la variable: `_X` par exemple.

Une telle solution rend le problème de la non identification univoque encore plus sérieux. Un procédé de numérotation peut néanmoins être utilisé, nous en parlerons plus bas.

2. Considérations d'implémentation

Trouver le nom que l'utilisateur a donné à une variable dans la clause courante ne pose pas de problème majeur. Nous allons analyser ici les problèmes que peuvent poser l'implémentation d'une solution dans laquelle on trace les variables libres en donnant le nom que l'utilisateur a donné à la variable dans la clause où elle a été créée.

Le réflexe habituel, face à la question de savoir comment implémenter une telle solution, est de se baser sur ce que l'on fait dans les langages traditionnels. Il s'avère qu'une table de symboles (qui à chaque adresse fait correspondre un nom de variable) n'est pas aussi facile à gérer en Prolog que dans un langage procédural classique.

Analysons les implications d'une telle méthode de manière plus approfondie.

Imaginons qu'à chaque création de variable, on lui donne un nom (que l'on cherche dans le texte source). On enregistre dans un tableau (éventuellement indexé...) la correspondance entre l'adresse de la variable et son nom. Chaque fois qu'il faut afficher la référence d'une variable, on cherche son nom dans la table à l'aide de son adresse. Remarquons qu'à un nom de variable dans le texte source correspondent plusieurs instances de variables à l'exécution et que l'appartenance d'une variable à une procédure ne peut pas être déduite de par sa position dans les piles.

Cette méthode nécessite:

- d'avoir un mécanisme qui permet de trouver le nom d'une variable dans le texte source lors de sa création. Dans le cas d'un interpréteur, cela est possible puisque l'on a déja des références vers le texte source pour

d'autres raisons. Dans le cas d'un Prolog compilé, les noms peuvent être préparés à la compilation.

- de mettre à jour la table des symboles après l'unification de chaque clause. C'est-à dire d'y ajouter de nouvelles correspondances adresse/nom.

- de mettre à jour la table des symboles lors des rétro-parcours. En effet, lors du rétro-parcours, des adresses enregistrées dans la table des symboles se libèrent suite à la destruction de la pile Prolog jusqu'au dernier point de choix.
  Une manière de résoudre le problème est d'implémenter la table des symboles sous forme de pile et d'enregistrer le sommet de la pile dans les points de choix. De cette manière, il est possible de restituer la pile dans son état initial lors du rétro-parcours.

Cela comporte des inconvénients non négligeables sur les performances, même lorsque l'on ne débugge pas. En effet, un test sur le mode d'exécution (mode d'exécution normale ou en mode debugging) est nécessaire après chaque unification, à chaque création de point de choix et à chaque rétroparcours. En mode de debugging, la rapidité est moins prépondérente, néanmoins, il faut éviter que l'exécution des parties de codes que l'on désire "skiper" ne consomme trop de temps.

L'intérêt d'avoir une table des symboles séparée du reste des piles est principalement de séparer l'implémentation de Prolog et celle de son debugger. Il se fait que, comme nous l'avons vu plus haut, des éléments propres à Prolog (l'unification et le rétro-parcours) rendent cette séparation impossible. Analysons dès lors une autre façon d'aborder le problème quitte à intégrer encore d'avantage le debugger et Prolog.

Deuxième solution: attacher un nom à chaque variable.

Dans un environnement, une variable est représentée sous la forme d'une cellule comportant un tag et une valeur. Le tag permet de savoir si la valeur est une référence vers un autre terme (ou vers elle-même si elle est libre), si c'est une constante, une liste, une structure,...

Le concepteur pourrait néanmoins décider d'ajouter un champ à la représentation d'une variable. Ce champ serait une référence vers son nom (vers le texte source du programme ou vers un élément d'une table de symboles créée à la compilation ou au chargement pour un interpréteur).

Cette méthode a l'avantage de ne nécessiter aucun traitement particulier lors du rétro-parcours. Et donc elle ne nécessite pas non plus d'informations supplémentaires à stocker dans les points de choix. La gestion de la table de symboles (qui est beaucoup plus petite et de taille fixe) est remplacée par la gestion du champ de référence vers une nom de variable. Le prix à payer est ce champ supplémentaire.

Enfin, une dernière méthode peut être utilisée si l'on décide de trailer les variables différemment.

Nous avons vu plus haut qu'une variable libre était représentée par un tag de variable et une valeur référençant la variable elle-même. Nous pourrions utiliser deux tags différents pour les variables: un tag pour une variable libre, un autre pour une variable liée. Cela a l'avantage de libérer le champ de la valeur pour d'autres utilisations: un nom de variable par exemple. En effet, le nom de variable ne nous intéresse que lorsque la variable est libre.

La méthode est donc similaire à la précédente mais le nom de la variable n'est présent que lorsque la variable est libre. Le problème est que le nom est perdu dès que

la variable est liée. Ce qui signifie que lors d'un rétro-parcours, l'untrailing sera incapable de rendre le nom de la variable à moins de l'avoir sauvé.

La solution est dès lors de ne pas sauver dans le trail uniquement les références des variables que l'on lie mais aussi la valeur qu'elles contenaient avant de les lier. Il s'agit d'un **trail "par valeur"**. Ce mécanisme double la taille de la pile de trail. Mais est une manière relativement propre de gérer les noms des variables.

3. Alternatives

Je vais citer ici quelques alternatives possibles au choix de représentation des variables qui permettent de résoudre le problème d'identification univoque des variables.

Une solution consiste à ajouter, derrière le nom de la variable, son adresse. Il s'agit en fait de la conjonction des deux méthodes décrites ci-dessus. Elle rend néanmoins les lignes de trace très longues et par là même moins lisibles.

Une solution intermédiaire consiste à concaténer au nom de la variable le numéro de la boîte d'invocation dans laquelle elle a été créée. Cette solution nécessite cependant de stocker tous les noms de variables concaténées avec leur numéro dans la table des symboles. Le texte du programme ne peut être utilisé pour référencer les noms. On est donc forcé d'utiliser la méthode de table de symboles dont la taille varie à l'exécution.

Une autre solution consiste à ajouter un numéro au nom de la variable. Chaque variable se voit associer un compteur. Ce compteur est incrémenté à chaque création d'une variable. D'un point de vue fonctionnel, c'est très confortable. Du point de vue de l'implémentation, il est évident que c'est relativement coûteux à gérer. La taille de la table reste cependant fixe et ne nécessite pas de gestion particulière au backtracking.

Enfin, une solution consiste à numéroter les variables seulement lors de l'affichage d'une ligne de trace si deux variables dénotent des entités différentes mais ont un même nom. La numérotation n'identifie la variable que pour cette ligne de trace. Ce procédé à l'avantage de garder de très petits numéros mais ne permet pas de voir l'historique d'une variable de manière univoque.

Le choix dépend des impératifs du Prolog à développer en opposant le confort de conception des programmes Prolog avec les impératifs de performance. Est-on prêt à ralentir légèrement l'exécution des programmes corrects (ou considéré comme tels) de manière à donner plus de confort à la phase de debugging? Est-on prêt à ralentir très sensiblement l'exécution des parties de programmes que l'on `'skip'` ? Enfin, est-on prêt à rendre impossible le debugging de certains programmes gourmands en place mémoire alors qu'ils tournent sans le debugger ? Ou en tournant le problème dans l'autre sens, est-on prêt à rendre la phase de debugging plus inconfortable pour des raisons de performance ? La décision dépend des buts que se sont fixés les implémenteurs du système.

4. Le problème des instanciations

Tel que je l'ai décrit, le modèle de la boîte pure produit les instanciations suivantes lors de l'affichage des valeurs instanciées des variables:

- une variable est toujours plus ou aussi instanciée dans une porte `EXIT` que dans la porte `CALL` ou `REDO` correspondante,

- une variable a toujours exactement les mêmes instanciations dans une porte `FAIL` que dans sa porte `CALL` correspondante,

- une variable a toujours exactement les mêmes instanciations dans une porte CALL que dans ses portes REDO correspondantes.

Les deux premiers points sont toujours vérifiés dans les implémentations. Pour des raisons techniques, d'autres choix peuvent néanmoins être faits pour les deux derniers points.

Il est logique que si l'on sort d'une boîte par la porte EXIT, on ne peut qu'avoir créé des liens (instancier des variables qui étaient précédement libres ou ne rien avoir instancié). On ne peut dans aucun cas avoir défait des liens qui existaient avant le CALL.

Lors du passage par la porte FAIL par contre, l'implémenteur du debugger pourrait faire plusieurs choix:

1. Montrer dans toutes les portes FAIL d'un rétro-parcours les instanciations du dernier point de choix. Cela consiste à untrailer[2] les variables qui ont été liées depuis le dernier point de choix avant de tracer le rétro-parcours. Ces instanciations sont plus anciennes que (ou identiques à) celles qui existaient lors du CALL correspondant (au FAIL que l'on trace).

2. Montrer dans toutes les portes FAIL d'un rétro-parcours les instanciations qui existaient lors de l'échec. Cela consiste à untrailer les variables qui ont été liées depuis le dernier point de choix seulement lorsque la porte REDO de ce point de choix sera traversée. Dans ce cas, les instanciations seront plus anciennes que (ou identiques à) celles de la porte CALL correspondante. Ce sera par ailleurs le cas pour toutes les portes REDO et FAIL qui seront traversées pendant le rétro-parcours

3. Montrer dans une porte FAIL les mêmes instanciations que celles de la porte CALL correspondante. Cela consiste à untrailer les variables qui ont été liées depuis le passage par la porte CALL correspondante.

Voici ce que donne les trois possiblités si l'on demande

```
?- p.
```

avec le programme de l'exemple 3.

Exemple 3:

[2] Défaire les liens qui ont été faits aux variables référencées dans le trail.

```
p :-
    liste_a_ou_b(Liste1),
    liste_a(Liste1),
    liste_a(Liste2),
    liste_abc([X,c,Y]).

liste_a_ou_b([Elément]) :-
    cst_a_ou_b(Elément).

liste_a([a]).

cst_a_ou_b(a).
cst_a_ou_b(b).

liste_abc([A,B,C]) :-
    cst_abc(A,B,C).

csts_abc(a,b,c).
```

La première possiblilité n'est jamais utilisée telle quelle, cela produit lors de l'affichage dans certaines portes FAIL des variables moins instanciées que dans la porte CALL correspondante.

```
(1)  0  CALL p
(2)  1  CALL liste_a_ou_b(Liste1)
(3)  2  CALL cst_a_ou_b(Elément)
(3)  2  EXIT cst_a_ou_b([a])
(2)  1  EXIT liste_a_ou_b([a])
(4)  1  CALL  liste_a([a])
(4)  1  EXIT  liste_a([a])
(5)  1  CALL liste_a(Liste2)
(5)  1  EXIT liste_a([a])
(6)  1  CALL liste_abc([X,c,Y])
(7)  2  CALL csts_abc(X,c,Y)
(7)  2  FAIL csts_abc(X,c,Y)
(6)  1  FAIL liste_abc([X,c,Y])
(5)  1  REDO liste_a(Liste2)
(5)  1  FAIL liste_a(Liste2)
(4)  1  REDO  liste_a([Elément])
(4)  1  FAIL  liste_a([Elément])
(2)  1  REdo liste_a_ou_b([Elément])
(3)  2  REDO cst_a_ou_b(Elément)
(3)  2  EXIT cst_a_ou_b(b)
...
```

La deuxième solution a l'avantage d'être facilement implémentable (de même que la solution 1) car la phase de untrailing est faite en une fois en utilisant les informations stockées (à cette intention) dans le dernier point de choix. Cependant, conceptuellement, beaucoups préfèrent voir dans le fail les mêmes instanciations que dans la porte d'entrée correspondante. Ils veulent voir dans la porte FAIL le but qu'il n'est pas (ou plus) possible d'établir. D'autres préfèrent voir dans la porte FAIL l'état des variables lors de l'échec (et donc avant d'untrailer). Ce choix n'est, à ma connaissance, implémenté dans aucun debugger. Il comporte l'avantage de montrer dans la porte FAIL jusqu'où l'exécution (l'unification) a pu être accomplie. C'est un choix très (voire trop) proche de l'implémentation.

```
(1)   0   CALL p
(2)   1   CALL liste_a_ou_b(Liste1)
(3)   2   CALL cst_a_ou_b(Elément)
(3)   2   EXIT cst_a_ou_b([a])
(2)   1   EXIT liste_a_ou_b([a])
(4)   1   CALL liste_a(Liste1)
(4)   1   EXIT liste_a([a])
(5)   1   CALL liste_a(Liste2)
(5)   1   EXIT liste_a([a])
(6)   1   CALL liste_abc([X,c,Y])
(7)   2   CALL  csts_abc(X,c,Y)
(7)   2   FAIL  csts_abc(a,c,Y)
(6)   1   FAIL liste_abc([a,c,Y])
(5)   1   REDO liste_a([a])
(5)   1   FAIL  liste_a([a])
(4)   1   REDO liste_a([a])
(4)   1   FAIL  liste_a([a])
(2)   1   REdo liste_a_ou_b([a])
(3)   2   REDO cst_a_ou_b(Elément)
(3)   2   EXIT cst_a_ou_b(b)
...
```

La troisième solution, quant à elle, nécessite de conserver la valeur du sommet de la pile de trail pour tous les noeuds de l'arbre. En effet, il faut pouvoir untrailer uniquement les liens qui ont été créés depuis l'entrée de la boîte (qui n'est pas un point de choix puisqu'il échoue). Tous les systèmes semblent faire ce choix. Nous verrons dans la section relative aux problèmes dus aux optimisations que cela n'est pas sans poser de problèmes.

```
(1)   0   CALL p
(2)   1   CALL liste_a_ou_b(Liste1)
(3)   2   CALL cst_a_ou_b(Elément)
(3)   2   EXIT cst_a_ou_b([a])
(2)   1   EXIT liste_a_ou_b([a])
(4)   1   CALL liste_a(Liste1)
(4)   1   EXIT liste_a([a])
(5)   1   CALL liste_a(Liste2)
(5)   1   EXIT liste_a([a])
(6)   1   CALL liste_abc([X,c,Y])
(7)   2   CALL csts_abc(X,c,Y)
(7)   2   FAIL  csts_abc(X,c,Y)
(6)   1   FAIL  liste_abc([X,c,Y])
(5)   1   REDO liste_a(Liste2)
(5)   1   FAIL  liste_a(Liste2)
(4)   1   REDO liste_a([a])
(4)   1   FAIL  liste_a([a])
(2)   1   REdo liste_a_ou_b(Liste1)
(3)   2   REDO cst_a_ou_b(Elément)
(3)   2   EXIT cst_a_ou_b(b)
...
```

En ce qui concerne les instanciations des variables dans les portes `REDO` (ou `REdo`), la même distinction peut être faite.

Voici les différentes possibilités:

1. Montrer dans la porte `REDO` (ou `Redo`) les instanciations du dernier point de choix.

2. Montrer dans la porte REDO (ou REdo) les instanciations de l'échec.

3. Montrer dans la porte REDO (ou REdo) les instatiations du CALL correspondant.

4. Montrer dans la porte REDO (ou REdo) les instanciations que l'on a montrées dans la porte EXIT correspondante.

Tout d'abord, je voudrais insister sur le fait suivant: une fois que des liens ont été défaits (suite à l'untrailing), il devient très difficile de refaire ces liens sans réexécuter les procédures qui avaient créé ces liens. En effet, les piles Prolog contiennent les variables auquelles les liens ont été appliqués et le trail contient des références vers ces variables. Si l'on défait ces liens, ni le trail ni les piles Prolog ne peuvent servir à recréer ces liens. Le fait est que refaire des liens que l'on a défait ne comporte aucun intérêt du point de vue de l'exécution du programme.

Si l'on veut être capable de refaire des liens que l'on a défait (sans réexécuter), il faudrait mémoriser, lorsque l'on défait les liens, non seulement les variables que l'on défait (elles étaient déjà mémorisées dans le trail) mais aussi les liens qu'elles contenaient avant de défaire ces liens. Le mot lien doit être compris ici au sens large: cela peut être un lien vers une autre variable, liste ou structure mais aussi la valeur d'une constante. Même un trail par valeur ne peut servir à cette fin car il enregistre les valeurs des variables avant de les instancier. Ce dont nous avons besoin ici, ce sont les valeurs avant de défaire les liens. Après étude de son coût, un tel mécanisme nous a semblé excessif.

C'est la raison pour laquelle ce qui suit préssuppose donc que nous ne pouvons que défaire des liens dans un rétro-parcours.

Dès lors, les choix faits pour les instanciations de la porte FAIL influencent ceux de la porte REDO.

Si l'on a choisi la méthode 1 ou 2 pour les portes FAIL, on est forcé de choisir respectivement les méthodes 1 ou 2 pour les portes REDO (ou REdo) puisque ces méthodes constituent en fait un choix sur le moment auquel il faut untrailer.

Si l'on choisi la méthode 1 ou 3 pour les portes FAIL, on ne peut plus choisir la méthode 2 pour les porte REDO ou REdo.

Nous pouvons aussi conclure qu'il est impossible de montrer dans une porte REDO les mêmes instanciations que dans la porte CALL correspondante (méthode 3). En effet, la troisième trace de l'exemple 3 montre dans la porte REdo de la boîte 2 des instanciations plus anciennes que celles montrées dans la porte REDO du dernier point de choix (la boîte 3). La ligne

```
(2)  1  REdo liste_a_ou_b(Liste1)
```

aurait dû être tracée

```
(2)  1  REdo liste_a_ou_b([Elément])
```

car lors du dernier point de choix, Liste1 a déjà été unifié à la liste [Elément].

Les choix que l'on fera concernant les instanciations dans les portes REDO ne sont pas entièrement dépendants de ceux que l'on fait à propos des portes FAIL. En effet, il est possible de montrer dans les portes FAIL non déterministes les mêmes instanciations que celles montrées dans la porte CALL correspondante puis, à partir de

la trace de la première porte REDO ou REdo que l'on rencontre sur le rétro-parcours, montrer les instanciations du dernier point de choix.

Cette méthode nous garantit (si l'on ne montre pas les REDO déterministes) que les instanciations des portes FAIL respectent toujours le point 3 tandis que les instanciations des portes REDO respectent le point 1.

Remarquons que ce choix (dirigé par des considérations techniques) n'entraîne pas forcément des instanciations étranges dans les portes REdo ni dénuées d'intérêt. Réentrer dans la boîte 2 par la porte REdo avec [Elément] plutôt que Liste1 signifie que la boîte 2 va d'abord chercher un autre solution en ne se donnant que Elément comme degré de liberté. Il se peut néanmoins que, suite à un rétro-parcours ultérieur interne à la boîte 2 (il n'y en aura pas dans ce exemple-ci), un degré de liberté plus grand soit envisagé.

Enfin, pour en terminer avec les instanciations, remarquons que le choix de Byrd a été d'utiliser la méthode 3 pour toutes les portes FAIL (deterministes ou non) et donc la méthode 4 pour les portes REDO et REdo. Cela signifie que les instanciations des portes FAIL sont toujours les mêmes que les instanciations de la porte CALL correspondante et que celles des portes REDO et REdo sont les même que celles de la porte EXIT correspondante. Les variables d'une porte REDO apparaissent dès lors plus (ou aussi) instanciées que dans la porte CALL correspondante.

5. Autres informations importantes (extensions)

1. Les points de choix

Il est souvent utile de savoir si une procédure est déterministe ou non de manière à prévoir plus facilement le rétro-parcours.

Il est dès lors utile de montrer à l'utilisateur si une procédure dont on sort est encore un point de choix ou non.

Cette information est d'autant plus importante si le Prolog effectue un indexage des clauses d'une procédure. En effet, dans ces conditions, une procédure contenant plusieurs clauses peut être quittée de manière déterministe dès le premier appel.

Une solution est de tracer différement la porte EXIT dans une situation déterministe que dans une situation non déterministe.

2. Le cut

Le cut va modifier le caractère déterministe de certaines procédures, il est donc important (pour les raisons données ci-dessus) que l'utilisateur sache exactement de quelles procédures il s'agit.

Le cut peut être considéré comme une procédure prédéfinie. Cela signifie qu'elle est tracée comme les autres procédures, avec une porte d'entrée (CALL) et une porte de sortie (EXIT). La procédure cut réussi toujours.

Les effets de bord de la procédure cut sont montrés en traçant toutes les boîtes qui étaient des points de choix et qui deviennent déterministes de par son exécution.

Une porte supplémentaire est introduite (porte CUT)par soucis d'homogénéité, elle ne correspond ni à une porte d'entrée ni à une porte de sortie; elle signale simplement que la boîte était anciennement un point de choix et qu'elle devient déterministe.

Les instanciations montrées sont les instanciations courantes (au moment du cut).

Exemple:

```
p(X):-
     a(X),
     b(X),
     !,          % signe du cut
     c(X).

a(a).
a(b).

b(X):-
     d(X).

d(a).
d(b).

c(b).
```

L'exécution de la preuve du but `p(X)` produit la trace suivante:

```
(1)  0   CALL p(X)
(2)  1   CALL a(X)
(2)  1  *EXIT a(a)  % asterisque = non detrministe
(3)  1   CALL b(a)
(4)  2   CALL d(a)
(4)  2  *EXIT d(a)
(3)  1  *EXIT b(a)
(5)  1   CALL !
(4)  2   CUT  d(a)
(2)  1   CUT  a(a)
(5)  1   EXIT !
(6)  1   CALL c(a)
(6)  1   FAIL c(a)
(1)  0   FAIL p(X)
```

3. Le contrôle de la trace

Lorsque l'on debug un programme, on a souvent une idée approximative de l'endroit où l'erreur devrait se trouver. Ce que l'on désire, c'est tracer pas à pas l'exécution en cet endroit. L'intérêt du débugger serait ramenée à peu de chose s'il fallait tracer l'exécution complète sans pouvoir contrôler le flux des informations. Je vais décrire dans cette section les différents mécanismes qu'il est utile d'avoir à sa disposition.

1. Atteindre l'endroit désiré (spy, leap)

Imaginons qu'une procédure d'un programme semble contenir un bug. L'utilisateur voudrait avoir la possibilité d'être informé de toutes les portes de cette procédure qui sont traversées. De la sorte, il peut plus facilement contrôler si les résultats rendus par la procédure sont cohérents par rapport aux questions posées et aux spécifictions de la procédure.

Ce mécanisme est offert par la plupart des debuggers. Il permet de placer un point espion (spy point) sur une (ou plusieures) procédures. Une exécution est alors lancée. Dès qu'une porte d'une boîte d'une procédure sur laquelle il y a un point espion est traversée, l'exécution est arretée et une ligne de trace est affichée. L'utilisateur reçoit alors le contrôle du debugger. Et peut, par exemple, demander de continuer l'exécution jusqu'au prochain passage par une telle porte.

Généralement, ce mécanisme met à la disposition de l'utilisateur les outils suivants:

- un prédicat qui permet de positionner un point espion sur une procédure ou une liste de procédures: `spy/1`,
- un prédicat qui permet de supprimer un point espion d'une procédure ou d'une liste de procédures: `unspy/1`,
- une commande du debugger qui permet de demander de continuer l'exécution jusqu'au passage au travers d'une porte d'une boîte d'une procédure sur laquelle il y a un point espion: `leap`.

Voici un exemple à l'aide des procédures de l'exemple 3. Les caractères en gras sont les caractères tapés par l'utilisateur, le reste est affiché par Prolog ou par le debugger.

```
?- spy(liste_a_ou_b/1).
debugger switch on, leap mode.
spy point set on Liste_a_ou_b/1.

yes.

?- p.
(2)  1  CALL liste_a_ou_b(Liste1) dbg- leap
(2)  1  EXIT liste_a_ou_b([a]) dbg- leap
(2)  1  REdo liste_a_ou_b([Elément]) dbg- leap
(2)  1  EXIT liste_a_ou_b([b]) dbg- leap

no.
```

2. La boîte noire (skip)

Lors du debugging de procédures récursives, les points espions peuvent s'avérer inefficaces. En effet, si l'on est par exemple intéressé par les résultats successifs fournis par une boîte donnée, on ne veut pas voir le détail de son exécution à la l'intérieur de la boîte, même si celle-ci s'utilise elle-même. La commande 'skip' donnée sur une porte d'entrée de boîte continue l'exécution jusqu'au passage au travers d'une porte de sortie de la même boîte.

Voyons cela sur l'exemple 4:

Exemple 4:

```
delete(X,[X|Xs],Xs).
delete(X,[Y|Ys],[Z|Zs]) :-
    delete(X,Ys,Zs).


?- spy(delete/3).
debugger switch on, leap mode.
spy point set on Liste_a_ou_b/1.

yes.

?- delete(a,[a,b,c],X).
(1)  0  CALL delete(a,[b,c,a],X) dbg- skip
(1)  0  EXIT delete(a,[b,c,a],[b,c]) dbg- leap

yes.
```

tandis que si l'utilisateur avait demandé un leap, il aurait reçu le détail de l'exécution à l'intérieur de la boîte 1:

```
?- delete(a,[a,b,c],X).
(1)  0  CALL delete(a,[b,c,a],X) dbg- leap
(2)  1  CALL delete(a,[c,a],Zs) dbg- leap
(3)  2  CALL delete(a,[a],Zs) dbg- leap
(3)  2  EXIT delete(a,[a],[]) dbg- leap
(2)  1  EXIT delete(a,[c,a],[c]) dbg- leap
(1)  0  EXIT delete(a,[b,c,a],[b,c]) dbg- leap

yes.
```

La commande 'skip' est la commande typique que l'on utilise sur une porte d'entrée lorsque l'on n'est pas intéressé par l'intérieur de la boîte, mais plutôt par le résultat.

Il peut être aussi utile de pouvoir signaler au debugger qu'une procédure donnée doit être toujours skippée (parce que l'on sait qu'elle est correcte par exemple). Cela consiste à considérer cette procédure comme une boîte noire.

On utilise pour cela les prédicats `skipped/1` et `unskipped/1` dont la syntaxe est la même que pour les prédicats `spy/1` et `unspy/1`.

3. Analyser l'exécution (creep, show var).

Une fois que le point du programme à debbuger a été atteint, l'utilisateur va vouloir analyser l'exécution pas à pas: la commande '`creep`'.

Chaque fois que l'utilisateur demande un '`creep`', l'exécution continue jusqu'au franchissement de la prochaine porte (peu importe s'il s'agit de la porte d'une procédure "espionnée" ou non).

Suite à l'exécution d'une procédure, des liens ont été faits à des variables créées avant l'appel de cette procédure. Il peut être intéressant de voir la valeur d'une variable donnée à un moment donné même si elle n'apparaît pas dans les arguments de la porte que l'on vient de franchir.

Cela nécessite évidement un mécanisme d'identification univoque d'une variable. Choisissons de les identifier à l'aide du numéro de boîte dans laquelle elles ont été créées[1].

Voici un exemple sur l'exemple 3:

```
?- delete(a,[a,b,c],X).
(1)   0  CALL delete(a,[b,c,a],X) dbg- creep
(2)   1  CALL delete(a,[c,a],Zs) dbg- creep
(3)   2  CALL delete(a,[a],Zs) dbg- show var: Z(2)
Z(2) = [c|Zs]
(3)   2  CALL delete(a,[a],Zs) dbg-[2] creep
(3)   2  EXIT delete(a,[a],[]) dbg- show var: Z(2)
Z(2) = [c]
(3)   2  EXIT delete(a,[a],[]) dbg- ...
```

4. D'autres déplacements rapides (goto, skip until var)

Après plusieurs exécutions d'un même programme à la recherche d'un bug, l'utilisateur finit par connaître par coeur le numéro de la boîte qui lui pose problème. Il peut utiliser la commande '`goto`' en spécifiant le numéro de boîte auquel il veut se rendre.

La commande '`goto`' peut aussi servir à continuer l'exécution jusqu'à un niveau donné. Cela est particulièrement utile pour descendre rapidement dans des sous-boîtes très imbriquées d'une procédure récursive.

Une syntaxe particulière permet de différencier un numéro de niveau avec un numéro d'invocation (des parenthèses par exemple).

---

[1] Il s'agit d'un cas idéal. Dans la pratique, la variable sera plutot identifiée par une addresse.

[2] nous avons choisis ici de répéter la dernière ligne de trace après la commande `show var`.

La commande 'skip until var' permet de continuer l'exécution jusqu'à ce que la variable spécifiée soit instanciée ou déinstanciée (lors d'un rétro-parcours).

5. Filtrage de l'information (backtrack skip, same port skip)

Comme je l'ai fait remarquer plus haut, il n'est pas toujours utile de voir tout le cheminement au travers des portes lors du rétro parcours. L'utilisateur aimerait donc pouvoir "skipper" toutes les portes (elles peuvent être très nombreuses dans des programmes fortement déterministes) qui mènent au dernier point de choix. C'est le but de la commande 'backtrack skip'.

L'utilité de la commande 'same port skip' est fort similaire. Elle continue l'exécution tant que la porte franchie est du même type. Dans le cadre d'un rétro-parcours, elle permet de "skipper" une série de portes FAIL ou une série de portes REDO. Elle permet aussi de descendre dans les sous-boîtes imbriquées (série de portes CALL) ou de sortir directement d'une procédure "tail recursif" (série de porte EXIT)

6. Le retry

La commande 'retry' permet de faire gagner énormément de temps lors du debugging. Il s'agit d'une facilité de debugging propre à Prolog car elle découle du mécanisme de rétro-parcour.

Lorsque l'utilisateur demande de faire un 'retry' d'une boîte donnée (par son numéro d'invocation), l'exécution reprend dans l'état où elle était lors de la création de cette boîte.

D'un point de vue technique, il s'agit d'un rétro-parcours mais sur la première clause de cette procédure.

D'un point de vue fonctionnel, le but est de recommencer l'exécution à un endroit donné du programme pour voir (ou revoir) une partie importante (pour l'utilisateur) de la trace.

Il est en effet fréquent de passer outre l'endroit du bug et de vouloir recommencer l'exécution de ce passage sans être obliger de réexécuter complètement le programme.

Cette commande permet également d'utiliser des statégies de debugging. Il permet par exemple de skipper les procédures (on passe de la porte d'entrée à la porte de sortie) de niveau 0 les unes après les autres jusqu'à ce qu'une d'entre-elles donne des résultats erronés. A ce moment, on demande un 'retry de cette procédure et l'on demande un 'creep'. On passe alors au niveau inférieur et l'on utilise la même méthode. De la sorte, on descend progressivement vers la sous-boîte (ou du moins une des sous-boîtes) qui fournit un résultat erroné.

7. Autres outils (fail, break, repeat)

La commande 'fail' permet de forcer un rétro-parcours, par exemple, parce que l'on n'est interressé que par les solutions engendrées par la seconde clause d'une procédure.

La commande 'break' permet de créer une nouvelle exécution de Prolog indépendante de l'exécution précédente.

Cela permet, par exemple, de faire un test d'une procédure, indépendamment du programme en cours de débugging, sans quitter cette session de debugging. Puis de revenir à cette session et de la continuer là où on l'avait abandonnée.

Certaines commandes telles que la commande `creep` sont répétées souvent. Il est fréquent de vouloir se rendre "une dizaine de portes plus loin" dans l'exécution. Ce mécanisme peut être offert à l'utilisateur en lui permettant de préciser le nombre de fois qu'une commande doit être répétée.

Par exemple '`10 creep`' est équivalent à taper 10 foix '`creep`'.

3. Implémentation du debuggeur

Un debugger est en fait une extension de la machine abstraite. De nouvelles instructions abstraites doivent être créées, de nouveaux registres sont utilisés et une nouvelle pile peut éventuellement être utiliser pour sauver les informations propres au debugger (c'est le cas que nous envisagerons dans ce qui suit). Différents couplages peuvent être envisagés entre la machine de base (sans débugger) et la machine avec debugger, nous aborderons également ce suget.

1. Choix pour Sepia[1]

1. Modèle de la boîte étendu & modèle de la boîte pure

Le modèle de la boîte pure telle que Byrd l'a définie est utilisée par un nombre important de systèmes Prolog du marché. Beaucoup d'utilisateurs y sont habitués et semblent s'en accomoder.

Dès lors, malgré les critiques que nous avons soulevé plus haut à son sujet, nous avons décider de proposer ce modèle[2] à l'utilisateur du debugger de Sepia en plus du modèle étendu (proposé par défaut) tel que nous l'avons décrit à la section 2.2.

Certains choix ont néanmoins été pris sur certains points. Ils restreignent les spécifications de la section 2.2. pour des raisons de performances. Les arguments de choix ont été les suivants:

- rendre le plus indépendant possible l'implémentation du debugger et celle de Prolog. Cela signifie entre autre qu'il faut toucher le moins possible (éventuellement pas du tout) au jeu d'instruction existant.

- éviter, dans la mesure du possible, qu'un programme (très gourmand en place mémoire) devienne "non débuggable" par suite de manque de mémoire lors de l'utilisation du debugger alors que le programme fonctionne sans problème sans le debugger.

- éviter, dans la mesure du possible, une trop grande différence de temps calcul entre l'exécution sans débugger et l'exécution avec debugger; cela afin d'éviter des temps d'attentes exessifs lors du '`skip`' de procédures[3].

2. Affichage des variables orienté implémentation

Nous avons décidé (dans un premier temps en tous cas) de nous contenter d'un affichage des variables orienté implémentation (identification des variables par leur adresse sur les piles).

---

[1] Sepia est un système Prolog compilé implémenté à l'ECRC (European Computer-Industry Research Centre) à Munich. Il implémente une machine abstraite comportant beaucoup d'extensions par rapport à la machine abstraite de Warren.

[2] avec la possiblilité de tracer la porte NEXT dans ce modèle aussi.

[3] L'implémentation actuelle du debugger provoque un ralentissement approximatif de l'exécution d'un facteur 6 (passer de 30 secondes d'attente à 3 minutes).

3. Trailing

Pour le modèle étendu, nous avons décidé de montrer dans les portes FAIL les mêmes instanciations que dans la porte CALL ou REDO correspondante et dans les portes REDO, les instanciations du dernier point de choix (instanciations lors du passage par la porte CALL de la boîte du dernier point de choix dont on va traverser la porte REDO).

Pour le modèle de la boîte pure, nous montrons les mêmes instanciations que celles de la dernière porte FAIL non déterministe.

4. Procédures compilées en mode non-debug et procédures déclarées skippées

Afin de permettre le debugging de programmes volumineux de manière incrémentale, le debugging de programme dont certaines procédures sont compilées en mode 'debug' (c'est-à-dire avec les invocations nécessaires au débugger incluses à la compilation) et d'autre compilées en mode 'non-debug' (c'est-à-dire sans invocation au débugger incluses dans le code) doit être rendu possible.

Etant donné le mode d'obtention de l'information (voir section 3.4) du debugger, une procédure compilée en mode 'debug' ne sera pas tracée lorsqu'elle est appelée à l'intérieur d'une procédure compilée en mode 'non debug'.

Les procédures compilées en mode 'non debug' sont en fait traitées comme des procédures déclarées skippées. Seuls les passages par leurs portes d'entrée et de sortie sont tracées. Il n'est pas possible de faire de `'creep'` à l'intérieur de telles procédures.

L'avantage est que lors de l'exécution de telles procédures, le debugger n'accomplit aucune action (pour les procédures non-debug) ou très peu (pour les procédures déclarées skippées). Cela entraîne à la fois un gain de temps considérable et une économie de mémoire importante car aucune information n'est conservée par le debugger pour les procédures internes à la procédure compilée en mode non-debug ou déclarée skipée.

5. Le Retry (problèmes et implications)

Afin d'éviter un éclatement des piles lors du debugging de procédures très récursives (mais malgré tout raisonablement déterministes), nous avons décider de conserver les optimisations des sous-arbres déterministes et de l'appel terminal.

Cela entraîne certaines complications mais aussi des restrictions. En effet, comme l'implémentation du `'retry'` est étroitement lié au backtracking, vouloir permettre d'accomplir un `'retry'` sur n'importe quelle boîte interdirait toute optimisation des sous-arbres déterministes et de l'appel terminal.

Nous avons donc décidé de ne pas permettre le `'retry'` des sous-boîtes d'une boîte de procédures déterministes hors de laquelle le flux de l'exécution est déja sorti.

De même, à cause de l'optimisation de l'appel terminal, on ne peut pas faire de `'retry'` d'une procédure dont le dernier sous-but a déja été appelé dans une situation déterministe car son environnement a déja été détruit.

Une autre restriction découle des effets de bord du cut. Il empêche les `'retry'` de toutes les boîtes à un niveau inférieur ou égal à celui du cut. Ce problème est détaillé plus bas.

2. Particularités de Sepia

Je vais décrire dans cette section quelques particularités de Sépia par rapport à la machine abstraite de Warren.

Malgré le fait que le jeu d'instruction de la machine abstraite de Sepia comporte un certain nombre de différences par rapport à celui de la machine abstraite de Warren, je vais expliquer l'implémentation du debugger dans le cadre de la machine de Warren de manière à rester le plus général possible[1].

Dans la suite, je vais faire abstraction du fait que la machine abstraite de Sepia est émulée par un émulateur écrit en C et que son jeu d'instruction est particulièrement plus étendu que celui de la machine abstraite de Warren.

Je vais supposer que l'on a, à tout moment, l'accès à tous les registres de la machine abstraite (y compris les registres d'arguments).

Certaines caractéristiques propres à Sepia seront néanmoins expliquées dans la mesure où ils peuvent guider des choix.

C'est le cas de l'organisation des piles de Sepia. Elle se différencie de celle de la machine abstraite de Warren par le fait que les environnements et les continuations ne sont pas stockés sur la même pile que les points de choix.

Les points de choix sont stockés sur une pile séparée: la **pile de contrôle**.

Le debugger de Sepia utilise une pile indépendante des autres piles de Prolog. Les raisons qui peuvent guider un tel choix seront détaillées plus bas. Cette pile peut contenir plusieurs types d'informations: des pointeurs vers les piles locale, globale, et de contrôle, des pointeurs vers la pile du debugger, des variables, des drapeaux, des foncteurs et arités de procédures (ou un identifiant d'un dictionnaire des procédures). Sa gestion sera détaillée au fur et à mesure dans les sections qui suivent.

3. Problèmes dus aux optimisations

1. Toutes les procédures ne sont pas des points de choix

Pour des raisons évidentes de performances, seuls les points de choix comportent des informations nécessaires au backtracking.

Dans la mesure où nous voulons pouvoir utiliser l'option `'retry'` sur des boîtes qui ne sont plus des points de choix, il faudra conserver quelque part cette information.

Soit on modifie le mode de fonctionnement du système en considérant toutes procédures comme points de choix, ce qui empêche toute optimisation. Soit on conserve cette information sur une pile propre au debugger, ce qui duplique de l'information.

Notons également que, même sans l'option `'retry'`, il y a des informations nécessaires au debugger qui ne sont présentes que dans le point de choix. Les arguments sont par exemple nécessaires pour pouvoir montrer les instanciations lors du passage par la porte `EXIT` et doivent donc être conservés au delà de l'appel.

---

[1] Et de ne pas dévoiler toutes les particularités d'une machine en cours de développement.

2. Optimisation de l'appel terminal

L'optimisation de l'appel terminal cause des problèmes principalement par le fait de l'écrasement des niveaux.

On peut, à ce propos, faire la différence entre le niveau logique (le nombre de boîtes imbriquées dans lesquelles on est entré) et le niveau physique (le nombre d'ancêtres que possède le noeud courant). Dans le cas d'une procédure écrite avec la récursion terminale, le niveau physique reste constant alors que le niveau logique augmente. Lors de l'exécution de l'instruction `proceed`, le niveau physique diminue d'un niveau alors que le niveau logique diminue de plusieurs en une fois.

Ce procédé implique la destruction d'informations devenues inutiles pour l'exécution mais importantes pour le debugger. Celui-ci doit conserver tous les noeuds au moins jusqu'à leur sortie.

Une autre problème concerne l'option '`retry`'. Lorsque le dernier but d'une procédure est appelé, l'environnement de la procédure est détruit. Cela n'empêche pas de faire des retry de la procédure mais bien de ses sous-buts: ils n'ont plus l'environnement nécessaire au fonctionnement des instructions de chargements des registres.

Nous avons dès lors décidé d'interdir de faire un '`retry`' d'une sous-boîte dont on a déjà appelé la dernière sous-boîte dans une situation déterministe.

3. Optimisation des sous-arbres déterministes

Cette optimisation ne pose pas de problème pour la trace dans le cadre du modèle étendu, le débugger peut même décider de ne pas conserver d'information concernant de tels noeuds une fois que l'on est sorti de la procédure. Il n'aura jamais besoin de cette information lors d'un rétro-parcours dans la mesure où le modèle étendu ne montre pas le rétro-parcours au travers de sous-arbres déterministes.

Il en pose cependant pour la trace du rétro-parcours des sous-arbres déterministes dans le cadre du modèle de la boîte pure. Mais dans la mesure où de toute façon le stockage d'informations propre au débugger était déja nécessaire à cause de l'optimisation de l'appel terminal, il ne s'agit pas d'un nouveau problème.

Il y a par contre un problème propre à l'optimisation des sous-arbres déterministes: l'impossibilité d'utiliser l'option '`retry`' sur une sous-boîte d'une boîte dont on est sorti de façon déterministe. En effet, lors de la sortie, sa continuation et son environnement ont été dépilées. Lorsque l'on restituera l'état de l'exécution de ce moment, le registe `E` va référencer une continuation qui n'existe plus. De même, les instructions d'unifications ne pourront plus fonctionner dans la mesure où l'environnement aura été dépilé.

Si nous supprimons cette optimisation, l'optimisation de l'appel terminal ne peut plus fonctionner (celle-ci nécessite le déterminisme de tous les sous-buts non teminaux).

Nous avons donc décidé d'interdir le '`retry`' sous-buts des procédures dont on est sorti par la porte `EXIT` de manière déterministe. Néanmois, l'utilisateur a encore le loisir de demander un '`retry`' de la procédure déterministe elle-même.

Remarquons enfin que comme la porte `EXIT` est tracée avant l'instruction proceed, l'utilisateur a encore la possibilité de faire le '`retry`' avant le passage par la porte `EXIT`.

4. Indexage

L'indexage ne pose qu'un problème mineur. Les notifications de rétro-parcours ne sont effectuées au debugger qu'avant les instructions `retry` et `trust`. Cela signifie que le rétro-parcours ne peut être tracé par le debugger que lorsqu'il y a un rétro-parcours phyique. Dès lors, un programmeur Prolog, qui n'est pas au courant de l'indexage, croit être en train de faire un REDO d'une clause (dont l'unification doit échouer) et voit l'unification réussir (car l'indexage a essayé directement une clause suivante) et un premier sous-but être appelé...

Une solution est de compiler sans indexage lorsque la compilation est en mode debug. Une autre est de prévenir l'utilisateur de l'existance de l'indexage.

Les arguments en présence sont les suivants:

- garder l'indexage augmente le déterminisme des programmes rendant l'optimisation des sous-arbres déterministes plus efficace et dès lors aussi l'optimisation de l'appel terminal. La supprimer augmenterait la différence de performence (aussi bien en place mémoire qu'en temps calcul) entre l'exécution avec et sans le debugger;

- moins un programme est déterministe, plus la trace est encombrée de rétro-parcours;

- l'utilisateur non averti peut être dérouté.

Le choix de Sepia a été de conserver l'indexage.

5. Tous les liens ne sont pas trailés

Le but du trailing est d'être capable de remettre le système dans l'état dans lequel il était lors de l'appel du dernier point de choix. Les algorithmes d'unification enregistrent donc, sur la pile de trail, les références de toutes les variables qui ont été liées suite à l'unification.

Il se fait que tous les liens n'ont pas besoin d'être trailés. En effet, les variables qui ont été créés après le dernier point de choix (qu'elles aient été allouées sur la pile locale ou sur la pile de recopie) sont apellées à disparaître lors d'un backtracking. Il est donc inutile d'enregistrer les modifications faites à ces variables dans le but de les restituer dans l'état où elles étaient lors du dernier point de choix si, de toute façon, elles n'existaient pas au moment de la création de ce point de choix.

Or, le debugger exige que tous les liens soient trailés pour deux raisons:

1. Nous voulons monter dans la porte `FAIL` les mêmes instanciations que dans la porte d'entrée correspondante y compris lors de l'échec de procédures déterministes.

2. Pour être capable d'implémenter l'option '`retry`' du debugger, nous devons être capable de faire un rétro-parcours même sur des noeuds qui ne sont pas des points de choix[1]. Si les variables ne sont pas restituables dans leur état initial, il est impossible de faire un rétro-parcours.

---

[1] Certaines restrictions existent néanmoins.

Cette optimisation est effectuée dans les algorithmes d'unification. Un premier réflexe serait donc de modifier ces algorithmes de façon à ce que toutes les variables liées soient enregistrées dans le trail.

Comme nous voulons éviter que la présence d'un débugger dans le système ait trop d'implications négatives sur la place mémoire utilisée et sur les performances de temps (untrailer des variables inutilement lors des rétro-parcours), nous allons ajouter un test dans les algorithmes d'unification de façon à ne faire l'optimisation que dans le cas ou le debugger est actif. Il se fait que l'algorithme d'unification est très souvent utilisé lors de l'exécution. Cela signifie que même lorsque le debugger n'est pas actif, l'exécution sera légèrement ralentie (un test supplémentaire à chaque lien de variable).

Pour certaines machines cet argument peut ne pas avoir de force. Certaines ont un registre (`SP` pour status pointer) qui permet de diriger l'exécution vers l'une ou l'autre routine d'unification selon le status de l'exécution (débugger actif ou non, mode coroutining ou non,...).

Une autre solution est de créer de nouvelles instructions abstraites. La solution est cependant coûteuse en nombre d'instructions suplémentaires. Des liens sont trailé aussi bien dans les instructions de chargement des arguments (pour les globaliser) que dans les instructions d'unification. Dans le cadre de Sepia, le problème est d'autant plus important qu'il possède déja un nombre important d'instructions d'unification.

Dans le cadre de la machine abstraite de Sepia, une autre solution existe. Elle découle du fait que, comme je l'ai signalé plus haut, les points de choix ne sont pas sauvés sur la même pile que la continuation et les environnements.

Il y a alors un registre supplémentaire: `EB` qui pointe vers l'environnement correspondant au dernier point de choix.

Voyons en détail le test qui est fait avant de trailer un lien dans le cadre d'une machine abstraite de Warren. La règle est la suivante: il ne faut pas trailer un lien à une variable à l'adresse `Adr` si `Adr` est supérieur (la pile locale et la pile globale fonctionne en adresses croissantes) à `B` pour une variable locale ou `HB` pour une variable globale.

Une possibilité serait donc de forcer les valeurs de HB à H et de EB au sommet de la pile de locale avant d'exécuter les instructions d'unifications (c'est-à-dire juste après un `allocate` et juste après un `call`) et de restituer ces valeurs avant un `call` ou un `proceed` afin de permettre le trimming et l'optimisation de l'appel terminal et avant un `proceed` pour permettre l'optimisation des sous-arbres déterministes.

Cela peut se faire par l'ajout de deux instructions dans le jeu de la machine:

- une instruction de forçage: `set_hb_eb` qui force `HB` et `EB` respectivement au sommet de la pile globale et au sommet de la pile locale.

- une instruction de restitution: `reset_hb_eb` qui restitue `HB` et `EB` aux valeurs sauvées dans le dernier point de choix de la pile de controle.

Cette solution n'est pas applicable dans une machine abstraite de Warren car si l'on force la valeur du registre B lorsque le point de choix n'est pas au sommet de la pile locale, un échec de l'unification va provoquer un rétro-parcours vers un noeud qui n'est pas un point de choix. La raison du problème est donc que le registre B sert à la fois de pointeur vers le noeud du dernier point de choix et de test d'ancienneté des variables par rapport au dernier point de choix. Dans la structure à deux piles (une pile locale et une pile de contrôle), EB ne sert pas au rétro-parcours, c'est `B` (qui pointe sur un bloc de contrôle) qui est utilisée.

4. Mode d'obtention de l'information et traitement

Dans cette section, je présuppose que toutes les procédures sont compilées en mode debug. Nous reviendrons sur le traitement des procédures compilées en mode non-debug dans une section ultérieure.

La machine abstraite communique avec le debugger à l'aide des portes. Cela signifie qu'elle notifie le débugger du passage du flux de l'exécution par une porte d'une procédure. Cette notification est en fait implémentée par une instruction abstraite supplémentaire (l'instruction abstraite `debug`) insérée, aux endrois adéquats, dans le code compilé.

Un des paramètres de l'instruction `debug` est le nom de la porte franchie. Les autres paramètres (qui dépendent de la porte franchie) sont détaillés dans les sections relatives à chaque porte.

Néanmoins, étant donné certaines optimisations, toutes les portes franchies ne sont pas signalées au debugger. Certaines d'entres elles doivent dès lors être déduites du contexte par le debugger.

1. Les portes CALL et CALL_LAST

Une instruction

```
debug CALL Proc
```

est insérée dans le code lors de la compilation avant les instructions `call`. Il notifie le debugger de l'appel d'un but en position non terminale.

Une instruction

```
debug CALL_LAST Proc
```

est insérée dans le code lors de la compilation avant les instructions `execute`. Il notifie le debugger de l'appel d'un but en position terminale.

Il est indispensable que le debugger soit notifié de manière différente pour l'appel d'un but en position terminale et pour l'appel d'un but en position non terminale. En effet, à cause de l'optimisation de l'appel terminal, le passage par la porte EXIT ne peut être notifié que pour les faits (l'instruction `proceed`).

L'optimisation de l'appel terminal fait "sauter" de niveau en dépilant les noeud de la pile locale et en mettant la continuation à jour avant la sortie (et même avant l'appel effectif) de la clause.

Pour être capable de tracer toutes les portes de sorties, le debugger doit dès lors mémoriser des informations que l'optimiation de la récursion teminale a détruite (la véritable sructure des appels c'est-à-dire l'arbre ET logique), le foncteur de la procédure appelée, les arguments ainsi que la propriété "terminale" ou "non terminale" de chaque appel.

Le block d'entrée peut aussi stocker les informations supplémentaires: le niveau logique de la boîte et son numéro d'invocation.

Afin de garder une indépendance maximum entre les informations nécessaires à Prolog et les informations nécessaires au debugger, nous avons décidé de sauver ces informations sur une pile séparée[1]. Le debugger empile dès lors un bloc contenant ces informations lors de chaque appel (CALL ou CALL_LAST). Nous pouvons considérer ce bloc comme une entrée de boîte. Nous l'appellerons le **bloc d'entrée**.

La pile devant représenter un arbre ET d'exécution, le bloc d'entrée sauve également un pointeur vers le bloc d'entrée de sa boîte mère.

2. La porte EXIT

Comme je l'ai dit plus haut, un passage par la porte EXIT ne peut être notifiée au débugger que pour les faits.

Une instruction

```
debug EXIT
```

est générée avant toute instruction proceed.

Lorsque le debugger reçoit la notification du passage par une porte EXIT, il doit déduire toutes les portes par lesquelles le flux de l'exécution est passé.

Pour cela, il utilise la propriété terminale des appels empilés. Si l'on sort du dernier sous-but d'une clause, on sort également de cette clause. Le debugger sort donc de toutes les boîtes dont le bloc d'entrée dont la boîte est terminale, y compris de la mère de celle-ci.

Nous verrons plus loin que pour pouvoir tracer les portes REDO non déterministes, les blocs d'entrée ne peuvent pas être dépilés lors de la sortie de la boîte (sauf si elle est déterministe dans le cadre du modèle étendu). Nous allons utiliser un pointeur vers le bloc d'entrée de la boîte la plus récente (la plus récement entrée) dont on n'est pas encore sorti. Dans la suite, nous qualifierons d'**ouverte** une boîte dont on est pas encore sorti.

Enregistrer la sortie d'une boîte consiste donc à mettre à jour le pointeur de la dernière boîte ouverte en le faisant pointer sur le bloc d'entrée de la boîte mère de la boîte dont il pointait le bloc d'entrée.

En plus de cette mise à jour, le débugger empile également un bloc de sortie, mais nous verrons son utilité plus tard.

3. Les portes TRY, RETRY et TRUST

1. Les notifications

Le debugger doit être notifié de la création d'un point de choix (porte TRY), d'un échec qui produit dès lors un rétro-parcours (porte RETRY et porte TRUST) et de la destruction d'un point de choix (porte TRUST)[2].

---

[1] D'autres éléments entrent en compte pour le choix de l'endroit où sauver les données du débugger, voir la section suivante à ce sujet.

[2] Le terme de porte est élargi ici à tout événement qui s'applique à une clause ou une procédure. La porte TRUST par exemple est à la fois une porte de clause qui signale un échec et une porte qui

L'instruction

```
debug TRY
```

est générée avant toute instruction `try` ou `try_me_else`.

L'instruction

```
debug RETRY
```

est générée avant toute instruction `retry` ou `retry_me_else`.

L'instruction

```
debug TRUST
```

est générée avant toute instruction `trust` ou `trust_me_else_fail`.

Ces invocations ne signalent au debugger ni la procédure qui a échouée ni celle vers laquelle on fait un rétro-parcours. Cela peut être déduit par le debugger.

La clause qui a échouée ne peut être que la clause courante de la dernière procédure ouverte. Si cette clause est la dernière "acceptable"[1] de la procédure, il y a un passage par la porte `FAIL` de la procédure. Sinon, il s'agit d'une porte `NEXT`.

2. Reconnaître les points de choix

Pour savoir quel est le bloc d'entrée de la procédure qui est le dernier point de choix, il y a principalement 2 méthodes.

La première consiste à utiliser un booléen, sauvé dans chaque bloc d'entrée qui signale si la boîte est un point de choix ou non. Appelons le **le booléen de point de choix**. Ce booléen doit être mis à jour lors du passage par une porte `TRY` (la boîte est un point de choix) et `TRUST` (la boîte n'est plus un point de choix). Cette méthode comporte le désavantage de ne pas être facilement utilisable. Pour savoir si une boîte est capable de faire un `REDO` non déterministe, il ne suffit pas d'analyser le booléen de point de choix du bloc d'entrée. Il faut aussi connaître les booléens de ses sous-boîtes. Afin d'éviter un parcours coûteux des blocs de la pile du débugger, nous pouvons utiliser un deuxième booléen (le **booléen de sous point de choix**) qui signale si une sous-boîte de cette boîte contient un booléen de point de choix.

La mise à jour des deux booléens est moins coûteuse que la consultation du booléen unique. Comme nous n'avons besoin du booléen de point de choix uniquement pour une procédure fermée (il n'est pas possible (voir modèle) d'avoir un `REDO` d'une procédure ouverte), celui-ci peut n'être mis à jour qu'à la sortie de la boîte. Il suffit dès lors de parcourir les boîtes filles sur un seul niveau. Si une d'entre-elles a un des deux booléens positionné, il faut positionnner le booléen de sous-point de choix.

---

signale le passage à la clause suivante d'une procédure tout en signalant qu'il s'agit de la dernière clause d'un groupe de clause.

[1] Dont le premier argument a un type compatible avec celui du registre A1 (voir l'indexage).

La deuxième méthode est beaucoup plus confortable. Il s'agit de chaîner les blocs d'entrée de la même façon que l'on chaîne les points de choix dans la machine abstraite de Warren. Cela nécessite un registre supplémentaire pour pointer le dernier point de choix. Un bloc d'entrée est celui d'un point de choix si le pointeur le référence, il possède un descendant qui est un point de choix si le pointeur référence un bloc d'entrée plus récent.

Cette méthode est néanmoins plus coûteuse: tout d'abord il nécessite la sauvegarde d'un pointeur en plus dans chaque bloc d'entrée, ensuite, il nécessite un registre supplémentaire. Cela peut être très important dans une machine "concrète".

Enfin, il est important de signaler qu'une clause peut avoir 2 points de choix dans la mesure ou l'indexage se fait à deux niveaux. C'est le cas par exemple des procédures dont une clause contient une variable comme premier argument et deux autres clauses contiennent la même constante comme premier argument. La gestion du booléen de point de choix est donc légèrement plus complexe. Une méthode peut être de prendre un entier qui peut prendre les valeur 0, 1 et 2. L'entier est incrémenté lors d'un `try` et dévrémenté lors d'un `trust`. Un entier non nul correspond à un booléen de point de choix à vrai.

3. Trouver les boîtes filles

Comme je l'ai déjà signalé à la section concernant le modèle de la boîte pure, les informations conservées sur la pile locale ne nous suffisent pas pour être capable de tracer les portes `REdo` sans être obligé d'effectuer un parcours coûteux de l'arbre ET. En effet, remonter les bloc d'entrées jusqu'à un point de choix ne nous permet pas de rencontrer toutes les boîtes par lesquelles on "ré-entre", ces boîtes ayant été créées avant le point de choix.

La recherche d'un point de choix consiste à parcourir l'arbre ET jusqu'au dernier point de choix créé (qui n'est pas forcément le premier rencontré sur ce parcours). La régle de choix de la prochaine boîte à parcourir est la suivante: se rendre chez la plus jeune fille, s'il n'y en a pas, se rendre chez la plus jeune soeur ainée et s'il n'y en pas, se rendre chez sa mère.

Dans la description des modèles, nous avons utilisé une chaîne de sous-boîtes de manière à être capable d'atteindre la sous-boîte la plus jeune d'une boîte donnée. Implémenter cela sur une pile ne serait pas raisonnable.

Une première méthode consiste à enregistrer dans chaque bloc d'entrée un poineur vers sa boîte fille la plus jeune et un pointeur vers la plus jeune de ses boîtes soeurs ainées. Comme nous avions déja un pointeur vers la boîte mère, nous avons tout ce qu'il faut pour effectuer un rétro-parcours tel que décrit ci-dessus.

Une deuxième méthode consiste à utiliser des blocs de sorties. Le principe est d'enregistrer sur la pile du débugger toutes les sorties de boîte dans l'ordre où on en sort. De la sorte, nous avons une véritable représentation des boîtes imbriquées sur la pile du debugger. Tous les blocs d'entrée et les blocs de sortie des boîtes filles d'une boîte donnée sont enregistrées entre son bloc d'entrée et son bloc de sortie.

En utilisant cette méthode, on peut alors effectuer le rétro-parcours en parcourant la pile du debugger en sens inverse de sa création jusqu'au bloc d'entrée d'un point de choix (ce sera le dernier). Lorsque l'on avance, on enregistre les entrées et les sorties de boîtes, lorsque l'on recule on "ré-entre" (`REDO`) dans une boîte en traversant sa sortie et on en "re-sort" (`FAIL`) en traversant son entrée.

Le bloc de sortie est implémenté sous forme d'un pointeur vers son bloc d'entrée correspondant (de manière à savoir à quelle entrée cette sortie correspond lors du rétro-parcours) .

4. La trace du rétro parcours

Sur le chemin du rétro-parcours, un bloc d'entrée peut décrire trois types de boîtes:

- une boîte de procédure qui est est le dernier point de choix (il s'agit de l'aboutissement du rétro parcours); elle n'a forcement aucun descendant qui soit un point de choix;
- une boîte de prodédure qui a un descendant comme point de choix (peut importe quelle soit un point de choix ou pas).
- une boîte de procédure qui n'est pas un point de choix et qui est plus ancienne que le dernier point de choix; elle n'a donc aucun descendant qui est un point de choix non plus.

Lors du rétro-parcours sur la pile du debugger (provoqué l' instruction debug `RETRY` ou debug `TRUST`), pour chaque type de boîte de procédure, on peut rencontrer deux types de bloc:

- un bloc d'entrée;
- un bloc de sortie.

Il y a donc 6 cas à envisager, ils correspondent chacun à une porte particulière du rétro-parcours:

1. Si le sommet de la pile du debugger est le bloc de sortie du dernier point de choix, nous somme en train d'entrer à nouveau dans la boîte du dernier point de choix, il s'agit d'un `REDO`.
   Dans le cas d'une trace dans le modèle étendu, dernier point de choix étant atteint, la pile du debugger peut être détruite jusqu'au bloc d'entrée pointé par le bloc de sortie du sommet de la pile, enregistrer que la dernière boîte ouverte est celle du dernier point de choix (mise à jour du pointeur vers la dernière boîte ouverte), tracer le passage par la porte `REDO` et le rétro-parcours s'arrête ici.
   Dans le cas d'une trace dans le modèle de la boîte pure, il n'est pas sûr que le rétro-parcours soit terminé, il faut encore parcourir l'éventuel sous-arbre déterministe de ce point de choix. Dès lors, on trace le passage par la porte `REDO`, on enregistre le fait que cette boîte est à nouveau ouverte et on dépile le bloc de sortie. Le rétro-parcours continue.

2. Si le sommet de la pile du debugger est le bloc d'entrée du dernier point de choix, nous nous trouvons à l'intérieur d'une boîte ouverte, du côté de son entrée[1]. Comme il s'agit d'un point de choix, le flux de l'exécution ne sort pas de la boîte, il y reste et essaye la clause suivante, il s'agit d'une porte (de clause) `NEXT`.

---

[1] Il s'agit de l'entrée lorsque l'on avance mais de la sortie si l'on recule. En terme de rétro-parcours, il s'agirait dès lors plutôt d'une sortie (porte FAIL).

On se contente de tracer le passage par la porte NEXT, le rétro-parcours est terminé.

3. Si le sommet de la pile du debugger est le bloc de sortie d'une procédure dont un descendant et un point de choix, nous avons affaire à une porte REdo.
   Dès lors, on trace le passage par la porte REdo, on enregistre le fait que cette boîte est à nouveau ouverte et on dépile le bloc de sortie. Le rétro-parcours continue.

4. La situation où le sommet de la pile du debugger est le bloc d'entrée d'une procédure dont un descendant est un point de choix est impossible, cela signifierai que l'on a dépilé l'entrée de boîte de point du choix. Or, le rétro-parcours doit s'arrêter sur celle-ci sur son entrée.

5. Si le sommet de la pile du debugger est le bloc de sortie d'une boîte qui n'est pas un point de choix et qui n'a pas de point de choix comme descendant, nous sommes en train d'entrer à nouveau dans une procédure qui constitue un sous-arbre déterministe, il s'agit d'une porte REDO déterministe.
   Dans le cas du modèle de la boîte pure, on trace le passage par la porte REDO, on enregistre le fait que cette boîte est à nouveau ouverte et on dépile le bloc de sortie. Le rétro-parcours continue.
   Dans le cas du modèle étendu, puisque l'on ne trace pas le rétro-parcours au travers de sous-arbres déterministes, rien n'est tracé, la pile est dépilée jusqu'au bloc d'entrée correspondant y compris celui-ci. Le rétro-parcours continue.

6. Si le sommet de la pile du debugger est le bloc d'entrée d'une boîte qui n'est pas un point de choix et qui n'a pas de point de choix comme descendant, nous sommes à la sortie (dans le sens du rétro-parcours) d'une boîte qui n'est pas un point de choix, il s'agit d'une porte FAIL déterministe (cela ne peut arriver que dans le cas du modèle de la boîte pure). On trace le passage par la porte FAIL et on dépile le bloc d'entrée. Le rétro-parcours continue.

5. La pile

1. Piles confondues ou piles séparées

Nous avons supposé jusqu'à maintenant que toutes les informations nécessaires au débugger étaient stockées sur une pile séparée.

Cela comporte un avantage important qui est la relative autonomie des deux processus: l'exécution du programme et le debugger.

Cependant lorsque l'on voit la quantité d'information stockée par le debugger (en plus de celle déjà stockée par l'exécution), on est tenté d'analyser la possibiliter d'intégrer les deux.

Outre le fait qu'il serait envisageable de créer deux jeux d'instrutions, un pour l'exécution avec debugger, l'autre sans, serait-il possible d'envisager l'utilisation des piles locales (ou de contrôle) pour sauver les informations nécessaires au debugger. C'est ce que nous allons envisager dans cette section.

1. Une seule exécution

Il pourrait être envisageable d'écrire une machine abstraite dans laquelle le debugger est complètement intégré. C'est en fait la solution qui consite à supprimer toutes les optimisations: toutes les procédures sont considérées comme des points de choix, les piles ne sont dépilées que lors du rétro-parcours et les noeuds de la pile locale contiennent aussi bien les environnement, la continuation et les informations nécessaires au rétro-parcours que les informations nécessaires au débugger.

Voici donc les arguments qui interviennent dans le choix d'une telle solution:

+ il s'agit de la solution la plus compacte et la plus efficace pour le debugger;

+ elle permet de forcer une réexécution de n'importe quelle boîte;

- aucune optimisation n'est effectuée (l'optimiation des sous-arbres déterministes peut néanmoins être conservée si l'on interdit la réexécution de telles procédures);

- il faut maintenir deux implémentations de Prolog en paralèlle (une sans optimisation mais avec un debugger et une autre sans debugger mais avec optimisations); le problème est surtout de garantir le même comportement dans les deux systèmes.

2. Deux exécutions séparées

C'est la solution antagoniste de la précédente. Elle consiste à conserver une exécution de Prolog qui effectue toutes les optimisations et à y intégrer (de manière plus ou moins forte) un debugger.

On peut considérer deux manières d'implémenter ce choix:

- stocker les informations du debugger sur une les piles Prolog (locale ou de contôle);

- stocker les informations du debugger sur un pile qui lui est propre.

Tout d'abord, remarquons qu'il n'est pas envisageable d'utiliser la pile locale. En effet, les blocks d'entrée du débugger ont une durée de vie supérieure à celles des environnements libérés par l'optimisation de l'appel teminal.

Analysons dès lors la possibilité de sauver les informations sur la pile de contrôle.

L'intérêt de cette solution (par rapport à une solution à piles séparées) est de partager entre le débugger et Prolog les informations des points de choix.

Toutes les informations du point de choix sont nécessaires pour les réexécutions (l'option `'retry'`). Le debugger utilise dès lors deux types bloc d'entrée, le premier est une extention du point de choix (pour sauver des informations propres au debugger: niveau logique, numéro d'invocation,...), le second est un block complet dans le cas d'une boîte qui n'est pas un point de choix.

Il n'est plus possible dans ces conditions de dépiler un point de choix lors de l'exécution de l'instruction abstraite `trust` ou `trust_me_else_fail` ni lors de l'exécution du cut (le point de choix ne peut qu'être mis à jour (la clause suivante est la

clause fail)). Cela diminue les opportunités de libération d'environnements pour les optimisations des sous-arbres déterministes et de l'appel terminal.

L'économie de place est principalement celle due au fait qu'il ne faut pas dupliquer l'information des points de choix. Notons que cette économie est également possible avec une architecture à pile séparée (les blocs d'entrée peuvent référencer un point de choix qui complète les informations qu'ils stockent) mais qu'elle comporte les mêmes implications concernant la mise à jour de la pile de contrôle pour les instructions `trust` et `trust_me_else_fail` ainsi que pour le cut.

Voici donc les arguments en présence:

+ gain de place mémoire;

+ pas de pile supplémentaire;

- gestion délicate de la pile de contrôle et sujette à modification (quid d'une modification de la gestion de la pile de contrôle sur la gestion du debugger);

- nécessité de modifier le comportement des instructions et algorithmes qui dépilent la pile de contrôle hors du rétro-parcours (`trust_me_else_fail`, `trust`, cut,...) entrainant des baisses de performance des optimisations;

3. Le choix pour SEPIA

Pour SEPIA, le choix a été d'implémenter le débugger sur une pile séparée des piles de Prolog. La raison principale était que, comme je l'ai dit plus haut, il fallait toucher le moins possible au jeu d'instruction existant et garder le maximum d'indépendance entre Prolog et le debugger.

6. Le cut

Le cut est implémenté, dans une machine abstraite de Warren, à l'aide de deux instructions abstraites. Une première (`savecut`) sauve la valeur du pointeur vers le dernier point de choix au tout début de l'exécution de la clause qui contient un cut, l'autre (`makecut`) met à jour le pointeur B avec la valeur sauvée.

Dans la mesure où l'exécution du cut va modifier le comportement du rétro-parcours, le debugger doit en être notifié (que l'on ait l'intention de tracer le cut ou non). Cette notification peut lui être faite par la génération de l'instruction

```
debug CUT
```

avant toute instruction `makecut`.

Il n'est pas nécessaire pour le debugger de connaître la nouvelle valeur de B car comme il ne fait pas d'optimisation de l'appel terminal dans sa propre pile, la dernière boîte ouverte est donc la boîte de la procédure qui a appelé le `cut`.

Mettre à jour la pile du debugger consiste dès lors à passer en revue tous les blocs d'entrée, un à un, depuis le sommet de la pile jusqu'au dernier bloc d'entrée ouvert. Chaque boîte rencontrée qui a le booléen de point de choix positionné doit être tracée (porte `CUT`). Les blooléens de point de choix et de sous-point de choix doivent être effacés pour toutes les boîtes rencontrés sur ce chemin. Comme la boîte de la procédure dont le cut est un sous-but est ouverte, il n'est pas nécessaire de mettre à jour d'autres booléens de sous-points de choix.

7. Le cut_to

Le prédicat `cut_to(X)` est un prédicat prédéfini qui permet de faire un cut jusqu'à une adresse donnée `X` de la pile (locale ou de contrôle selon l'architecture de la machine). Il fonctionne en corrélation avec le prédicat `get_cut(X)` qui permet de sauver dans la variable `X` l'adresse courante du registre `B`.

Cette instruction est utilisée de manière interne (c'est-à-dire qu'il n'est pas disponible tel quel pour l'utilisateur) mais pourrait être utilisé pour implémenté des prédicats de contrôles particuliers lors d'extentions ultérieures du langage.

De même que pour le cut, il est indispensable que le débugger soit notifié de son exécution.

Le plus pratique est de notifier le debugger après l'exécution du cut_to car dans ce cas, la valeur de `B` a déjà été modifiée et il n'est pas nécessaire de communiquer la valeur de `X` au debugger (elle est dans le registre `B`).

Si ce prédicat est toujours compilé en mode debug, il suffit d'ajouter un test au traitement de la notification de la porte `EXIT`.

Sa gestion est légèrement plus délicate pour le debugger que celle du cut. D'une part il faut parcourir la pile du debugger bloc d'entrée par bloc d'entrée afin de tracer les portes `CUT` des points de choix et mettre à jours les booléens de points de choix et de sous-points de choix des blocs d'entrées dont la valeur de `B` sauvée est supérieur (si la pile locale ou de contrôle fonctionne en adresse croissante) à `X`.

D'autre part, il faut mettre à jour les booléens de sous-points de choix des procédures dont des descendants sont rendus déterministes par l'exécution du `cut_to`. Le problème ne se posait pas pour le cut car la boîte mère du cut étant ouverte, tous ses ancêtres sont forcément ouverts. Le cut_to quant à lui peut rendre déterministe des boîtres fermées (sans rendre déterministe ses ancêtres).

Afin de minimiser la longueur du parcours des blocs d'entrée dont le booléen de sous-point de choix doit encore être mis à jour, on peut utiliser les propriétés suivantes (soit `M` la plus ancienne des boîtes dont le booléen de point de choix a été mis à jour):

1. la valeur du booléen de sous-point de choix ne doit être à jour que pour les boîtes fermées;

2. comme toutes les boîtes de procédures qui sont rendues déterministes par le cut_to ont déjà eut leur booléen de point de choix mis à jour, seules les boîtes plus anciennes doivent être mises à jour (c'est-à-dire plus anciennes que `M`);

3. seules les boîtes de procédures qui sont des ancêtres de `M` doivent encore être mises à jour En effet, si la boîte (soit X) est plus récente que M elle a déjà été mise à jour (de par le point 2); si X est plus ancienne que M et qu'elle n'est pas un ancêtre de M, aucun de ses descendants n'ont été modifiés car alors ce descendant serait plus jeune que M (par le point 2), ce qui est absurde).

Cela mène à l'algorithme suivant: partir de la boîte mère de M et remonter ses ancêtres un à un jusqu'à une boîte ouverte ou une boîte qui doit garder son booléen de sous-point de choix. Chacune des boîtes rencontrées (exeptée celle qui est ouverte) doit être mise à jour. Mettre une boîte à jour consiste à effacer son booléen de sous-point de choix si aucune de ses sous-boîtes n'a un des deux booléens (de point de choix et de sous point de choix) positionné.

8. Le retry

1. Le principe

Le principe de l'option '`retry`' consiste exécuter un backtracking à la boîte spécifiée mais plutôt que de recommencer avec la clause suivante, on recommence la procédure avec la première clause.

Si le debugger sauve dans les blocs d'entrée toutes les informations d'un point de choix lors de l'instruction `debug CALL`, il faut en fait sauter à l'adresse `P` sauvée + `1`. Dans ce cas, l'instruction `call` va être réexécutée également.

Dès lors, il faut sauver la valeur du registre `P + 1` dans le bloc d'entrée plutôt que l'adresse de la clause suivante (tel que le font par exemple les instructions `retry(_me_else)` et `trust(_me_else_fail)`.

L'algorithme est donc similaire à celui du `fail`:

- toutes les variables enregistrées dans le trail depuis TR(bloc d'entrée) sont rendues libres
- les arguments A1 à Ai sont chargés avec les valeurs sauvées dans le bloc d'entrée
- les registres E, CP, TR et H sont restaurés avec les valeurs sauvées dans le bloc d'entrée
- le registre P est chargé avec l'adresse de l'instruction call sauvée dans le bloc d'entrée
- le registre B est restitué avec la valeur sauvée dans le bloc d'entrée
- la pile du debugger est dépilée jusqu'au bloc d'entrée (qui devient le sommet).

2. Les boîtes qui ne peuvent pas faire l'objet d'un 'retry'

Plusieur types de boîtes ne peuvent pas faire l'objet d'un 'retry'. Le problème vient d'une part des optimisations, d'autre part des effets secondaires du cut.

Il y a quatres cas à développer:

- les pertes d'informations dues au trimming;
- les pertes d'informations dues à l'optimisation des sous-arbres déterministes;
- les pertes d'informations duent à l'optimisation de l'appel terminal;
- les effets de bord du cut.

1. Le trimming

Avant l'appel à un sous-but (et après les instructions de chargement des arguments), des variables de l'environnement qui ne sont plus nécessaires pour les sous-buts suivants sont libérées. Cela a comme conséquence qu'il n'est plus possible de forcer un rétro-parcours sur certains sous-buts de la procédure car les prochaines instructions de chargement des registres risquent de travailler sur des variables de l'environnement qui n'existent plus.

Supprimer le trimming pose peu de problèmes et n'occasionne pas une augmentation démesurée de la taille de la pile locale.

Supprimer le trimming consiste simplement à toujours indiquer comme paramètre de l'instruction `call` la taille maximum de l'environnement.

2. l'appel terminal

L'optimisation de l'appel terminal est la phase finale du trimming: ce qui reste de l'environnement est libéré ainsi que la continuation. Cela n'est bien entendu possible qu'en situation déterministe, c'est à dire si aucun des sous-buts précédents n'ont créés de point de choix.

Comme nous l'avons déjà remarqué plus haut, cette optimisation rend impossible un rétro-parcours sur un des sous-buts d'une clause dont ont vient d'appeler le dernier sous-but.

Comme nous voulons garder l'optimisation de l'appel terminal, nous allons interdir le 'retry' d'une sous-boîte d'une boîte dont la dernière sous-boîte a été appelée en position déterministe.

Cette situation doit être détectée par le debugger lors d'une notification du passage par une porte `CALL_LAST`; toutes les sous-boîtes soeurs sont alors marquées de manière à signaler qu'il est impossible de faire un 'retry' sur cette boîte. Le dernier sous-but, par contre, peut être réexécuté puisqu'il n'a plus besoin d'environnement (l'instruction `debug call` se trouve après les instructions de chargement).

3. Les sous-arbres déterministes

L'optimisation des sous-arbres déterministes, quant à elle, supprime l'environnement lors de la sortie d'une boîte. Il est encore possible de réexécuter la boîte (son environnement est créé à nouveau) mais il n'est plus possible de réexécuter ses sous-boîtes.

La détection de la situation peut être faite à posériori. Une boîte ne peut pas être réexécutée si sa boîte mère est fermée avec un booléen de sous-point de choix non positionné.

4. Le cut

Le cut a des effets de bord que le rétro-parcours ne peut pas défaire. Cela rend la réexécution impossible pour toutes les procédures dont un ancêtre a été "coupé".

Observons le problème sur un exemple:

exemple:

```
p:-
     vaut123(X),
     vaut24(X),
     !,
     sous_but.

vaut123(1).
vaut123(2).
vaut123(3).

vaut12(1).
vaut12(2).

sous_but.
```

Voici la trace de la preuve du but `p` jusqu'à l'appel de `sous_but`:

```
(1)  0  CALL p
(2)  1  CALL vaut123(X)
(2)  1  EXIT vaut123(1)
(3)  1  CALL vaut24(1)
(3)  1  FAIL vaut24(1)
(2)  1  REDO vaut123(X)
(2)  1  EXIT vaut123(2)
(4)  1  CALL vaut24(2)
(4)  1  EXIT vaut24(2)
(5)  1  CALL !
(2)  1  CUT  vaut123(2)
(5)  1  EXIT !
(6)  1  CALL sous_but
```

Il est impossible, à ce stade de demander une réexécution de la boîte numéro 3 car le point de choix de la boîte 2 ayant été supprimé (et ne pouvant être restitué par un

rétro-parcours), le CALL de vaut24(1) provoquerait un échec sans possibilité de rétro-parcours sur vaut123(X).

Cela nous empêche de réexécuter une boîte dont un ancêtre a été coupé car la restitution de l'état ne peut être complet que si l'on recrée les points de choix coupés.

Les algorithmes du cut et du cut_to doivent dès lors marquer les blocs d'entrée des boîtes qu'ils coupent. Le debugger doit interdir l'option 'retry' sur une boîte dont la boîte mère a été coupée.

9. Les procédures compilées en mode non-debug

Une procédure compilée en mode non-debug ne contient dans son code aucune instruction de notification pour le debugger. Celui-ci n'est donc pas notifié de ce que la procédure compilée en mode non-debug appelle une autre procédure, du fait que sa porte EXIT a été traversée ni du fait qu'un rétro-parcours a eut lieu sur cette procédure.

Il est également possible qu'une procédure non-debug (ou procédure compilée en mode non-debug) appelle une procédure debug (ou procédure compilée en mode debug). Le debugger sera par contre notifié des appels de la procédure compilée en mode debug.

Le manque d'information généré par les procédures non-débug peut mettre le debugger dans des situations inextricables; il est absolument nécessaire de mettre en oeuvre des mécanismes adéquats "d'espionnage" de l'exécution de ces procédures.

L'intéret d'accepter le mélange de procédures compilées en mode debug et en mode non-debug au sein d'un même programme lors de son debugging, est de permettre de debugger des programmes volumineux. Les procédures correctes sont compilées en mode non-debug et leur exécution utilisera un minimum de place pour le debugger. Seul son appel génèrera un bloc d'entrée et sa sortie un bloc de sortie; les appels imbriqués à la procédure seront ignorés par le débugger. La trace ne pourra reprendre qu'à la sortie de la procédure non-debug.

Son exécution sera dès lors presqu'aussi performante qu'une exécution sans debugger autant au point de vue de la place mémoire utilisée qu'au point de vue du temps calcul.

1. Le bloc d'entrée

Pour que le debugger soit capable de traiter différement les procédures debug des procédures non-debug, il lui faut un outil capable d'obtenir le mode de compilation d'une procédure appelée (instruction debug CALL d'une instruction non-debug).

La compilation gère un dictionnaire des procédures. Ce dictionnaire doit enregistrer le mode de compilation de chaque procédure. Cette information doit être accessible au debugger.

Lors de l'appel d'une procédure non-debug, un bloc d'entrée est créé sur la pile du debugger. Ce bloc est marqué comme étant celui d'une procédure non-debug.

Tant que le sommet de la pile est un bloc d'entrée d'une procédure non-debug, le debugger sait que l'exécution est encore dans la boîte de la procédure non-debug. Il ignore dès lors les portes CALL ou EXIT qui lui sont notifiées (elles appartiennent à des procédures debug qui ont été appelées à l'intérieur de la procédure non-debug).

2. La porte EXIT et la porte TRY d'une procédure non debug

Par définition, la sortie de la procédure non-debug ne peut pas être notifié par une instruction insérée dans le code. Il nous faut un mécanisme qui va nous assurer que la sortie de la procédure non debug nous sera notifiée et ce mécanisme ne peut être mis en oeuvre que dynamiquement.

Une solution consiste donc à agir sur la continuation. Accomplir cela consite à modifier dynamiquement l'arbre d'exécution.

Soit `proc_nd` une procédure non-debug et `proc_d` la procédure debug qui l'appelle:

```
proc_d :-
     proc_nd, ....
```

est transformé dynamiquement en:

```
proc_d :-
     appel_nd,....

appel_nd :-
     proc_nd, exit_nd.
```

où `appel_nd` correspond à un appel traité dynamiquement par le debugger et `exit_nd` est compilé comme:

```
debug EXIT_ND
proceed
```

L'idée est de sauver une continuation supplémentaire sur la pile locale lors de l'appel de la procédure non-debug. Cette continuation référence les instructions qui permettent de notifier le debugger de la sortie de la procédure non-debug.

Comme on le voit sur la transformation ci-dessus, la continuation supplémentaire doit être empilée après celle de la procédure non-debug. Si on laisse l'instruction abstraite `call proc_nd` s'exécuter, la prochaine instruction sera celle du début de l'instruction `proc_nd` et la continuation (le registre `CP`) ne pointera pas vers la routine de notification. L'instruction `call proc_nd` doit donc être simulée par l'instruction `debug CALL proc_nd`[1]:

elle consiste à

[1] Ceci constitue un argument important en faveur d'une implémentation du debugger par l'ajout d'un jeu d'instructions abstraites à utiliser pour le debugger. Quid par exemple d'une implémentation où les instructions `call` ou `execute` ont des tailles variables dans le code; un analyse dynamique du code par le debugger est-elle une technique sure ?

- faire le traitement habituel du debugger concernant les portes CALL (bloc d'entrée,...)
- empiler sur la pile locale l'environnement courant (E) et (plutôt que CP) une continuation pointant sur l'instruction qui suit l'instruction call proc_nd
- charger dans CP l'adresse de la routine de notification de sortie.
- charger dans P proc_nd

Lorsque l'appel à la procédure non-debug se trouve en position terminale, la simulation (faite par l'instruction `debug CALL_LAST`) est bien sur celle de l'instruction `execute`:

- faire le traitement habituel du debugger concernant les portes CALL_LAST(bloc d'entrée,...)
- empiler sur la pile locale l'environnement courant (E) et la continuation (CP)
- charger dans CP l'adresse de la routine de notification de sortie
- charger dans P proc_nd

Lorsque le debugger est notifié du passage de l'exécution par une porte `EXIT_ND`, il doit, en plus d'effectuer les actions habituelles pour une porte `EXIT`, mettre à jour le booléen de point de choix.

En effet, la procédure non-debug ne notifie pas le debugger de l'exécution des l'instructions `try` ou `try_me_else`. Il faut dès lors mettre le booléen de point de choix à jour lors de la sortie de la procédure. Il doit être positionné si la valeur courante de `B` est supérieure (si la pile de contrôle ou locale fonctionne en adresse croissante) à la valeur de `B` sauvée dans le bloc d'entrée de la procédure non-debug; cela signifierai que un ou plusieurs points de choix ont été créés entre temps.

3. La porte `FAIL` d'une procédure non-debug

Comme pour les procédures debug, le passage par une porte `FAIL` est déduite par le debugger du fait du passage par une porte `RETRY` (avant les instructions `retry(_me_else)` ou `trust(_me_else_fail)`).

Plusieurs cas sont à envisager lorsque le sommet de la pile du debugger est un bloc d'entrée d'une procédure non-debug.

Deux critères permettent de les différencier:

- le dernier point de choix peut être une procédure debug ou non-debug;
- le rétro-parcours peut provoquer l'échec de la procédure non-debug (dont le bloc d'entrée est au sommet de la pile) ou uniquement de procédures appelées par la procédure non-debug.

Le cas du rétro-parcours lorsque le dernier point de choix est une procédure compilée en mode non-debug est développé à la section suivante, il nécessite la mise en oeuvre d'un mécanisme particulier.

Envisageons le cas où le dernier point de choix est une procédure debug. Dans ce cas nous sommes sûr que le debugger sera notifié du passage par la porte `RETRY`.

Les deux cas à envisager sont donc les suivants: soit le rétro-parcours est interne à la procédure non-debug (la porte RETRY ou TRUST est celle d'une procédure debug appelée par la procédure non-debug) soit le rétro-parcours sort de la procédure non-debug (passage par la porte FAIL). Nous sommes dans le premier cas losque le dernier point de choix est plus jeune que le point de choix référencé par la valeur de B sauvée dans le bloc d'entrée. Dans ce cas, la notification est simplement ignorée. Sinon, le traitement est celui d'une porte RETRY ou TRUST telle que nous l'avons déjà décrit plus haut.

4. La porte RETRY/TRUST

Comme une procédure compilée en mode non-debug ne possède aucune instruction debug, le debugger n'est pas notifié d'un rétro-parcours vers une procédure non debug.

Si la procédure non debug dont on franchit la porte RETRY ou TRUST est encore ouverte (on est pas encore passé sa porte EXIT), le rétro-parcours doit de toute façon être ignoré car il ne concerne que des boîtes internes à la procédure non-debug (ou éventuellement la porte NEXT de la boîte de clause de la procédure non-debug).

Le problème est donc de notifier le debugger du passage du flux de l'exécution par la porte RETRY ou TRUST d'une procédure non debug dont on est sorti.

De manière intuitive, résoudre ce problème consiste à transformer dynamiquement l'exécution du programme en ajoutant un point de choix (que je qualifierai **d'artificiel**) après la sortie d'une procédure non-debug.

Soit le programme suivant où proc_nd à été compilé en mode non debug:

Exemple:

```
proc_d :- proc_nd, sous_but.
```

Lorsque le débugger est notifié de la sortie du flux de l'exécution de la procédure non debug proc_nd et que proc_nd est un point de choix, l'exécution de la procédure sera modifiée dynamiquement en:

```
proc_d :- proc_nd, porte_retry_nd, sous_but.
```

où la compilation du prédicat porte_retry_nd est:

```
        try_me_else label_RETRY
        proceed
label_RETRY:
        trust_me_else_fail
        debug RETRY_ND
        fail
```

D'une manière pratique, la création du point de choix est accomplie par le debugger lors de la notification d'une porte EXIT_ND uniquement si la procédure non-debug est un point de choix (voir porte TRY). L'adresse de la clause suivante que référence ce point de choix est celle de la routine suivante:

```
        trust_me_else_fail
        debug RETRY_ND
        fail
```

dont le but est de dépiler le point de choix artificiel, de notifier le debugger du passage de l'exécution vers la porte RETRY ou TRUST du dernier (véritable) point de

choix (en l'occurence, celui d'une procédure non debug) et de faire un rétro-parcours (vers le véritable point de choix).

Le debugger ne fait pas de différence entre la porte RETRY et la porte TRUST, le booléen de point de choix ne peut servir qu'une foix la procédure non debug fermée, tant que la procédure est ouverte, le registre B peut être utilisé pour savoir si la procédure contient un point de choix ou non (voir porte FAIL)

5. Trouver le sommet de trail correct

Comme rien n'est empilé sur la pile du debugger pendant l'exécution d'une procédure non-debug, la valeur du registre TR sauvée dans le bloc d'entrée de la procédure non-debug n'est pas forcement celle qu'il valait lors de la création du dernier point de choix. Rappelons que le booléen du point de choix n'est mis à jour que par suite du changement de la valeur du registre B entre l'appel et la sortie de la procédure non debug.

Or, comme nous l'avons vu plus haut, lors de la trace des portes REdo, il faut untrailer jusqu'au sommet qu'avait la pile de trail lors du dernier point de choix. Untrailer jusqu'à un sommet plus ancien provoquerait des problèmes.

La valeur de TR à utiliser dans une porte REdo d'une procédure non debug doit donc être recherchée dans le dernier point de choix. Tandis que la valeur à utiliser dans une porte FAIL est prise dans le bloc d'entrée de la procédure.

6. Une limite au mélange des modes de compilation

Nous allons montrer dans cette section que le choix fait de ne rien tracer à l'intérieur d'une boîte d'une procédure non debug, même pas les procédures debug qui sont appelées par celle-ci, n'est pas uniquement un choix fonctionnel.

Voyons le problème sur un exemple:

Exemple:

```
proc_nd :-
     proc_d,
     !,
     sous_but.

proc_d.
proc_d :-
     sous_proc_d.

sous_but :-
     fail.

sous_proc_d.
```

Remarquons tout d'abord que si nous voulons être capable de tracer les procédures debug appelées par les procédures non debug, nous devons ajouter une notification de la porte call comme première instruction de la compilation d'une procédure.

En effet, si dans l'exemple ci-dessus proc_nd à été compilée en mode non debug et proc_d à été compilée en mode debug, l'appel de proc_d par proc_nd ne pourra être notifié au debugger que si proc_d contient une instruction du type

```
debug CALLED proc_d
```

comme première instruction de la procédure.

D'autre part cela n'est pas possible pour "l'appel" du cut dans la mesure où il s'agit d'une instruction (`make_cut`) insérée dans le code compilé de la procédure `proc_nd` qui, étant une procédure compilée en mode non debug ne peut contenir aucune instruction `debug`.

Lors de l'exécution de l'exemple ci-dessus, le cut ne sera pas notifié au debugger qui ne pourra dès lors pas mettre ses booléens de point de choix à jour.

Cela signifie que le prochain échec (celui de `sous_but` dans l'exemple), notifié au débugger par une notification (`RETRY` ou `TRUST`) d'une procédure plus ancienne que `proc_nd`, sera considéré par le debugger comme un rétro-parcours vers la procédure `proc_d`.

Notons enfin que le problème ne se pose pas si l'on ignore tout ce qui est appelé par une procédure non debug car un cut ne détruit pas de point de choix plus vieux que la procédure qui l'appelle (la procédure non debug elle même).

7. Le problème du `cut_to`

Contrairement à ce qui se passe avec le cut, le `cut_to` peut détruire des points de choix plus vieux que la procédure qui l'appelle.

La solution est de toujours compiler la procédure `cut_to` en mode debug.

Un dernier problème subsiste: le `cut_to` peut détruire un point de choix artificiel d'une procédure non debug sans détruire tous les points de choix de cette procédure.

La solution consiste à empiler un nouveau point de choix artificiel si la dernière procédure qui est un point de choix est une procédure non debug et que certains de ses points de choix ont été détruits. Ce test doit être fait lorsque le `cut_to` a déjà été exécuté (lors du passage par la porte `EXIT`)

Conclusion

Les principaux problèmes et solutions (que nous avons présenté dans la section 3) relatifs à l'implémentation du debugger concernaient le cas d'un debugger que l'on implémente sur un système Prolog existant que l'on désire modifier le moins possible. Les problèmes qu'une telle situation occasionne succitent les deux remarques suivantes.

D'une part, puisque ajouter un debugger sur un Prolog existant pose tant de problèmes, il semble impératif d'envisager le mode d'implémentation du debugger en parallèle avec l'analyse de l'implémentation du Prolog. En effet, c'est à ce stade qu'il faut faire une série de choix: la représentation des variables doit-elle prévoir un champ supplémentaire pour le nom des variables ? Le trail doit-il fonctionner par référence uniquement ou également par valeur ? Un registre (drapeau) de status d'exécution (débugger actif ou non actif) doit-il être prévu dans la machine abstraite ? Faut-il prévoir un mécanisme de maintenance efficace de deux jeux d'instructions ?...

D'autre part, si implémenter un modèle de trace séparé de l'implémentation est si couteux, ne serait-il pas intéressant d'analyser l'utilité d'un modèle très proche de l'implémentation - et même influencé par les optimisations ? En effet, puisque ces optimisations existent, n'est-il pas important que le programmeur en soit concient et constate ses résultats sur la trace ? Cette optique (qui est l'antagoniste de celle prise dans ce travail) comporterait certains désagrément car l'arbre d'exécution est constamment modifié par les optimisations mais l'idée mérite d'être approfondie.

En ce qui concerne l'extension du modèle de Byrd, le découpage de la boîte peut encore être poussé plus loin: on peut imaginer de séparer la tête de la clause et ses sous-buts. Le problème est de montrer les informations pertinentes sans encombrer la trace.

L'analyse fonctionnelle faite dans ce travail s'est axé sur le problème de la trace. Des couches supérieures peuvent être envisagées, c'est le cas, par exemple, des méthodes de debugging automatisés tels que les algorithmes de Shapiro qui permettent de localiser certaines erreurs de manière interactive. On peut aussi envisager d'utiliser Prolog comme langage de commande du debugger. Cela permet d'écrire des programmes Prolog qui traitent la trace (donc l'image de l'exécution) d'un autre programme. Ce sujet fait l'objet de recherche à l'ECRC dans l'équipe OPIUM.

Bibliographie

[ANL 85] J. Gabriel, T. Lindholm, E.L. Lusk, R.A. Overbeek, *A Tutorial on the Warren Abstract Machine for Computational Logic*, ARGONNE NATIONAL LABORATORY, 1985

[Boizumault 88] P. Boizumault, *PROLOG l'implantation*, colleciton ERI, edit. Masson, 1988.

[Bruynooghe] M. Bruynooghe, *The Memory Management of Prolog Implementations*, dans Logic Programming, K.L. Clark et S.A. Tarnlund, Academic Press, pp 83-98, 1982

[Byrd 79] L. Byrd, *The Prolog-10 debugging package,* D.A.I., University of Edimburg, 1079.

[Byrd 80] L. Byrd, *Understanding the Control Flow of Prolog Programs*, *Logic Programming Workshop*, Debrecen, 1080.

[Dufresne 88] P. Dufresne, *Sepia Emulator: Implementation Notes*, Internal Report IR-LP-13-02, ECRC, juillet 1988.

[Macartney 88] G. Macartney, *An Introduction to SEPIA*, Technical Report TR-LP-29, ECRC, février 1988.

[Meier 88a] M. Meier, *SEPIA Abstract Machine*, Internal Report IR-LP-13-03, ECRC, février 1988.

[Meier 88b] M. Meier, *The Implementation of Coroutining in SEPIA*, Technical Report IR-LP-13-00, ECRC, février 1988.

[Meier, van Rossum 89] M. Meier, E. van Rossum, *The specification of SEPIA Debugger*, Internal Report IR-LP-13-15, ECRC, fevrier 1989.

[SEPIA 88] M. Meier, G. Macartney, P. A. Tsahageas, D. Henry de Villeneuve, David Dhan, *SEPIA Version 2.0 User Manual*, Technical Report TR-LP-38, ECRC, fevrier 1988.

[Warren 83] D.H. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI Intrenationnal, Menlo Park, octobre 1983.